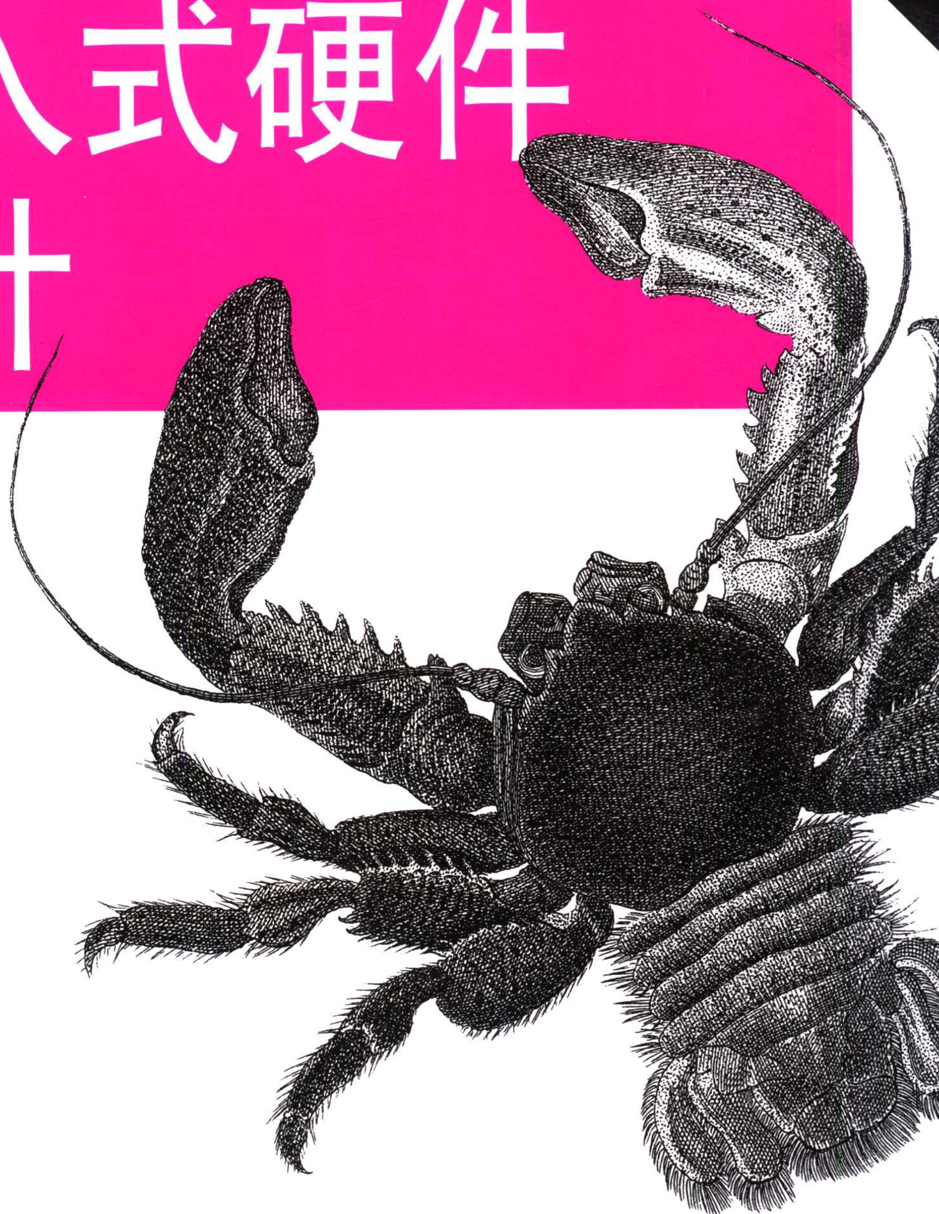
Designing Embedded Hardware

第二版

嵌入式硬件设计





John Catsoulis 著

O'Reilly Taiwan 编译

徐君明 陈振林 郭天杰 改编

O'REILLY®

中国电力出版社

第二版

嵌入式硬件设计

John Catsoulis 著
O'Reilly Taiwan 公司 编译
徐君明 陈振林 郭天杰 改编

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

嵌入式硬件设计：第2版/（美）卡特索利斯（Catsoulis, J.）著；徐君明，陈振林，郭天杰改编．－2版．－北京：中国电力出版社，2007

书名原文：Designing Embedded Hardware, second edition

ISBN 978-7-5083-5372-2

I. 嵌 ... II. ①卡 ... ②徐 ... ③陈 ... ④郭 ... III. 硬件－设计 IV. TP303

中国版本图书馆CIP数据核字（2007）第045133号

北京市版权局著作权合同登记

图字：01-2006-4877号

©2005 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2007. Authorized translation of the English edition, 2005 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2005。

简体中文版由中国电力出版社出版2007。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

书　　名/ 嵌入式硬件设计（第二版）

书　　号/ ISBN 978-7-5083-5372-2

责任编辑/ 牛贵华

封面设计/ Emma Colby，张健

出版发行/ 中国电力出版社（www.infopower.com.cn）

地　　址/ 北京三里河路6号（邮政编码100044）

经　　销/ 全国新华书店

印　　刷/ 北京丰源印刷厂

开　　本/ 787毫米×980毫米　16开本　23.5印张　527千字

版　　次/ 2007年10月第一版　2007年10月第一次印刷

印　　数/ 0001-4000册

定　　价/ 39.80元（册）

敬告读者

本书封面贴有防伪标签，加热后中心图案消失

本书如有印装质量问题，我社发行部负责退换

版权所有　翻印必究

O’Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O’Reilly Media, Inc. 授权中国电力出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc. 是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User’s Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O’Reilly Media, Inc. 一直处于Internet 发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O’Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media, Inc. 依靠他们及时地推出图书。因为 O’Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O’Reilly Media, Inc. 知道市场上真正需要什么图书。

作者简介

John Catsoulis 生活在澳大利亚布里斯班的热带阳光之下。他以量子物理、电子学和数学三门主修专业优异的成绩获得澳大利亚国立格林菲斯大学的理学学士学位，并获得澳大利亚拉托斐大学专用计算机体系结构专业的工学硕士学位。他曾经设计过的计算机系统多到他本人也记不清了，囊括了手指大小的机器到多处理器的计算机引擎。全世界许多公司和政府机构都用过他的设计和软件。John 还在大学任教，传授他在计算机体系结构和设计方面的心得。他目前正在昆士兰大学主持“太空飞行器容错可重构计算机”的研究。

不在发热的微处理器旁辛勤工作的时候，John 喜欢徒步旅行、露营、为野生动植物和风景摄影、钓鱼、灌溉和栽培、烹饪印第安地区和地中海的食物，以及同他的外甥 Andrew 和 James 一起玩模型火车。

封面介绍

本书封面上的小动物是瓷蟹。这种微小的无脊椎动物一般躲在沿太平洋海岸的潮池里，外壳黄褐色，躯干也只有 5 毫米长。瓷蟹有六对小腿，其中很小的一对蜷于尾巴的底部。尽管它们能够游弋于水中，但它们步行足末端那突起的锐利的刺能使其非常容易地紧紧抓住浸于水中的坚硬岩石的表面。瓷蟹腿上的绒毛可以粘附海底的泥土，有助于它们伪装食肉动物。从贻贝堆、海绵以及海藻也是它们的避难所。一旦瓷蟹藏匿于这些较佳的栖息地，就可以摆动着它们那柔臂游弋于水中，捕捉浮游生物和其他微小的动植物。当受到食肉动物的威胁时，瓷蟹就丢掉一条腿或爪来分散攻击者的注意力，这样狡猾的瓷蟹便可以逃之夭夭了，而它们所丢掉的附肢最终也会再长出来。

目录

前言 1

第 1 章 计算机体系结构介绍 7

概 念 8

存储器 21

输入 / 输出 25

DMA 25

嵌入式计算机体系结构 31

第 2 章 汇编语言 34

寄存器 36

机器码 36

有符号数 38

寻址模式 39

用汇编语言编程 41

反汇编 44

位置无关代码 45

循 环 45

屏蔽 46
索引寻址 47
堆栈 48
指令的时序 50

第3章 Forth与公开固件标准 52

Forth简介 52
字符串 55
堆栈操作 56
创建新词 58
注释 60
if…else 61
循环 62
数据结构 65
与硬件和存储器的交互 67
Forth程序设计准则 69

第4章 电子学概览 70

电压和电流 70
模拟信号 72
功率 73
理解电路原理图 73
电阻 78
电容 85
RC电路 87
电感 91
变压器 94
二极管 95
晶体 98
数字信号 102

电气特性 104
逻辑门 112
阅读技术手册的重要性 113

第 5 章 电源 114

来自壁上插座的电流 114
电池 115
低功耗设计 115
稳压器 116
LM78xx 稳压器 118
MAX603/MAX604 稳压器 120
MAX1615 稳压器 121
MAX724 稳压器 121
电气噪声与干扰 123

第 6 章 搭建硬件平台 127

工具 127
焊接 132
快速的构建方式 138
印制电路板 142
制作电路板 154
JTAG 158

第 7 章 用 SPI 添加外部设备 160

串行外围设备接口 160

第 8 章 用 I2C 添加外部设备 174

I²C 简介 174
使用 I²C 接口添加一个实时时钟 178
使用 I²C 接口添加一个小显示设备 179

第9章 串口 180

通用异步收发器（UART）... 180

错误检测 182

历史悠久而可靠的RS-232C 183

RS-422 190

RS-485 192

第10章 IrDA 196

IrDA 简介 196

第11章 USB 203

USB 简介 204

USB 包 206

物理接口 208

USB 接口的实现 211

第12章 网络 214

CAN 214

以太网 218

第13章 模拟量 226

放大器 226

模/数转换 230

连接外部ADC 233

温度传感器 235

光电传感器 237

加速计 240

压力传感器 241

磁场传感器 244
数/模转换 245
脉宽调制 247
电机控制 248
控制大负载 255

第14章 PIC微控制器 257

两款处理器的发展史 257
一个简单的实例 259
一个更大的PIC处理器 262
基于PIC的环境数据记录器 263
用PIC来控制电机 269

第15章 AVR微控制器 275

AVR处理器的体系结构 276
ATtiny15处理器 278
代码的下载 285
更强大的AVR处理器 287
基于AVR的数据记录器 289
总线接口 289

第16章 68HC11 314

68HC11的体系结构 314
一台基于68HC11的简单计算机 315

第17章 MAXQ 325

MAXQ概览 325
电路原理图 327

第 18 章 68000 系列计算机 332
68000 处理器的体系结构 333
简单的基于 68000 的计算机 337

第 19 章 基于 DSP 的控制器 346
DSP56800 系列 349
基于 DSP56805 的计算机 352
JTAG 360

前言

[启迪] 沉浸在数字计算机电路里的惬意之感，
就如同站在高山之巅或匿于花瓣之中的那种感觉一样。
——罗伯特 M. 波西格
《万里任禅游》

欢迎阅读《嵌入式硬件设计》第二版，希望本书能让你了解计算机硬件的构建过程。正如编写优良的软件中有亮点一样，一个设计精良的硬件也会有闪光之处。就嵌入式计算机而言，你要在各个层次上对其进行了解，同时也要掌握通过电线的电流驱动以及软件执行的复杂算法。事实上，如果没有对硬件的理解，就不可能编写出嵌入式软件；同样，没有对软件的理解，也不可能设计出硬件。你潜心于这一机器，以达到超过对台式机的了解的程度，最重要的是，这其中充满了乐趣。

本书在选择芯片和设计布局上，有意选择了那些实现起来价格不高而又极其有用的器件。我与本书中提到过的任何公司均没有商业、金融或其他方面的一些关系。之所以使用他们的芯片，仅仅是基于我自身经验上的个人偏好罢了。这些公司生产的芯片使用起来容易，是值得信赖的，也是健壮的，并且具有良好的技术支持，提供了完整而全面的技术资料。简而言之，本书所选的芯片更适合初学者。

在本书的第一版发行时，我特意不谈软件，原因有二：首先，市面上已经有很多书在探讨C语言程序设计、如何用C语言开发嵌入式软件、移植Linux到嵌入式系统、Python程序设计以及Java程序设计，等等（当然，O'Reilly所出版的相关书籍依然是其中最棒的）；其次，汇编语言是最基本的编程工具，然而，不同的处理器之间汇编语言的差异如此之大，以至于一本书的篇幅绝不可能完整涵盖每种微处理器的所有指令集，既然如此，我只有一视同仁了。但是，我在这一版里却决定纳入若干软件的内容。我并不打算

在本书中提及每种处理器的指令集，我只想介绍若干简单的汇编语言技术。尽管不同的微处理器体系结构之间汇编语言的差异如此之大，然而基本的概念却是一样的。

本书新增了一章来专门介绍Forth程序语言。Forth是一种相当老的编程语言，早已不再是主流的程序设计语言，因此你很难找到这方面的书籍。Forth是一个非常有用的嵌入式系统开发工具，遗憾的是许多工程师迄今尚不清楚此事。Forth是Open Firmware标准的依据，也是Apple、Sun和许多其他制造商的开发工程师们所采用的语言。Forth是相当有用的编程语言，值得你花点时间去学习一下。

本书中的多数设计看起来简单，事实上也确实如此。这些设计可以看作是简单的积木，你可以对其进行混合和搭配以实现你所需的嵌入式系统。我希望本书对你有所帮助。看完本书，自己动手做一做看看吧！

——John Catsoulis
澳大利亚布里斯班
2005年1月
jtc@embedded.com.au
http://www.embedded.com.au

本书的结构

本书大致分为四个部分。第一部分包括了一些基本概念和介绍性的资料。第二部分介绍了汇编语言以及Forth。有了这些背景知识后，将接着探讨计算机的外部设备以及如何为嵌入式系统添加新功能。最后，介绍被广泛应用于嵌入式系统的各种微处理器，以及把它们集成到计算机中的设计过程。

第1章介绍了计算机体系结构的概况，讨论了组成嵌入式系统的基本要素。第2章和第3章探讨了汇编语言和Forth。

第4章讲述了一些电子学理论的背景知识，介绍了一些重要的概念。如果你对电子学很精通的话就直接跳到第5章，这一章讲述了如何向嵌入式系统提供电源，以及如何保护好你的嵌入式计算机免受电气干扰和其他能够引起故障的小毛病的干扰。在第6章介绍了如何具体地构建并调试一个嵌入式计算机系统。

第7章和第8章介绍了SPI和I^2C，这是两种允许把大量小型外部设备添加到微控制器中的协议。第9～11章讲述了串行接口，这些接口使嵌入式系统可以接入主机以及诸如调制解调器（Modem）这样的外部设备。同时还讲述了RS-232C、RS-422、红外通信以及USB。

有关网络的部分将在第12章谈到，在这一章里你可以看到如何向嵌入式系统中添加低价位的工业网络（CAN）。也可以在该章中学到如何向嵌入式系统中添加以太网端口，通过这一端口，嵌入式系统可以连接到其他计算机、服务器和网关上，进而藉由这些设备访问互联网。

第13章涉及到了现实生活中的接口。在这一章里，将学到如何处理模拟信号和数字值之间的相互转换，如何在嵌入式系统里通过传感器来测量温度、光、电压、加速度和磁场，也可以学到如何使用一个嵌入式计算机来控制小型电机。

第14章是本书探讨微处理器部分的开始，你将认识第一个嵌入式微处理器体系结构：Microchip公司的PIC。在后续各章里，你会看到各种微处理器，从极小的单一8位微控制器，到32位基于总线的具有相当计算能力的微处理器。由于不可能涵盖每一种嵌入式微处理器（毫不夸张地讲，将有数百种之多），所以本书只好从中选取具有代表性的芯片。尽管如此，你在本章中所学到的技巧仍可应用到你为应用所选的任何处理器上。

代码示例的使用

本书用于帮助读者完成自己的任务。一般而言，你可以在自己的程序和文档中使用本书中的示例。你不用事先联系我们，除非是要进行大量复制。例如，在你写的程序中用到几段本书的程序代码，就不用经过我们的同意。但是，如果要刻录成光盘发布或销售来自O'Reilly书中的示例，就必须获得授权。回答别人的问题时，引用本书的文字和程序，也不用经过我们的同意。但在你的产品文件中若要大量加入本书的文字和程序，则必须获得授权。

虽非必要，但我们十分赞赏你在引用本书的内容和示例时提到出处。完整的引用信息包括书名、作者、出版商，还有ISBN。例如："Designing Embedded Hardware, by John Catsoulis. Copyright 2005 O'Reilly Media, Inc., 0-596-00755-8."

如果你觉得对代码示例的使用超出了合法使用的范畴或超越了所获取的许可权限，就请与出版商联系：*permiissions@oreilly.com*。

排版约定

本书采用的约定如下：

正文

```
Source code
```

Signal（高电平有效）

$\overline{\textbf{Signal}}$（低电平有效）

十六进制数字由 0x 前缀指明。

二进制数字由 % 前缀指明。

大写 K 为 1024，小写 k 为 1000。

联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市海淀区知春路 49 号希格玛公寓 B 座 809 室（邮编：100080）
奥莱理软件（北京）有限公司

本书的网页上列出了勘误表、示例和任何额外的信息。可登录以下网址查询：

http://www.oreilly.com/catalog/dbhardware2
http://www.oreilly.com.cn/book.php?bn=978-7-5083-5372-2

如果想要发表关于本书的评论和技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

有关本书的评论或问题也可以直接给作者发送电子邮件联系：

jtc@embedded.com.au

关于图书、会议、资源中心和 O'Reilly 网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

在此要特别感谢我的两位编辑 Andy Oram 和 Mike Hendrickson。我还要感谢第一版的编辑Jon Orwant。过去，我经常在前言中看到作者对出版社的编辑所给予的帮助如何地表示感谢，可是直到现在我才理解编辑的这种帮助所具有的深度和重要性。

正如你早已熟知的，O'Reilly 出版优秀的图书。我也要对本书的制作团队的大力协助表示谢意 —— 尤其是 Sanders Kleifeld。他们为本书所做的贡献不亚于我撰写本书时所做的努力。我还要感谢 David Chu 的所有协助。

感谢 Dallas Semiconductor、Kathy Vehorn、Don Loomis、Mike Quarles 以及 Moe Rubenzahl 的协助，让我得以取得 MAXQ 处理器预先发行的样片。谢谢 Peter Paine、Donna Mack 和 Cooper Tools、Karen Rolan 和 Fluke Corporation 以及 Tektronix 允许我在书中使用他们产品的照片。还要感谢 Picochip 公司 Rupert Baines 的协助。

Geoff McDonald 是我很要好的朋友，他在本书的内容上给了我很多有益的建议，他也为本书进行了校对，我对他的所有帮助表示感谢。

我还要感谢 Michael、Marry、Renee 以及 Michell Lees。Michael 帮助审阅了本书的书稿，并提供了许多有用的建议。

我要对 Jeff O'Keefe 博士这么多年来的友谊和支持表示谢意。从大学时代起，我们就成为朋友了。在大学二年级的实验室里，我们曾烧毁了电路板，也曾做过用光照射老师的事情。

感谢Neil Bergmann教授、John Williams博士、Keith Ball、Denis Bill、Barry Bettridge，以及昆士兰州立科技大学电气电子系统工程院的全体人员所给予的帮助。

我要感谢我的朋友和同事：David Nicholls、Peter Stewart、Mark Gentile、John Devlin 教授、Richard Wiltshire、Michelle 和 Robert Salier、Addy 和 Derek Clark、Kam Tam、Phil McDonald 以及 Vamsi Madasu。

最后也是最重要的，我要感谢我的大家庭所给予我的关爱和支持，尤其是我的妹妹 Kris 和我的两个小外甥 Andrew 和 James，他们的关爱和幽默使得生活充满乐趣。我也要对 Chris 和 Jeff Goopy 以及堂兄妹 Theo 和 Maree、David 和 Jenevieve、Michael、Andrew 和 Karen、Antony、Fiona、Drew、Ashley 和 Max 一直给予的友谊和帮助表示感谢。特别要感谢我的叔父 Vince 和 Dave Catsoulis，他们让我体会了爱、荣誉和力量这些品质的真谛。我十分感激他们。

第1章

计算机体系结构介绍

每一台机器都有其自身独特的个性，

这种个性或许可以定义为你对其所知所感的直观总和。

机器的这种个性经常改变，通常是越变越差，

但有时也会出乎意料地变好……

——罗伯特 M. 波西格

《万里任禅游》

这是一本有关设计和构建专用计算机的书。我们都知道计算机是什么样子 —— 它就是在你的桌案上单调地咕噜咕噜叫（如果风扇坏掉了则咔哒咔哒地响）的箱子。这个箱子运行着你的程序并且经常崩溃（如果你没有运行某些版本的Unix操作系统）。在这个箱子里面有运行你的软件、存储你的信息并把你与外界连接起来的电子设备。所以，设计一台计算机就是设计一台保存和操作数据的机器。

计算机系统基本上分为两类。第一类就是常见的台式计算机，当你跟别人说“计算机”的时候，别人所理解的通常就是这样一台机器。第二类是嵌入式计算机，这种计算机往往为控制和/或监控目的而集成到另外的系统中。嵌入式计算机在数量上远远超过桌面系统，而这乍看起来很不明显。随便问一个人他家里总共有多少台计算机，他会回答家里有那么一两台。实际上，他家里可能有30台计算机，甚至更多，这些计算机可能藏在电视机、录像机、DVD播放器、遥控器、蜂窝电话、烤箱、玩具以及其他一些设备里面。

在本章中，我们将着眼于通常的计算机体系结构。这种体系结构对嵌入式计算机和台式计算机都是适用的，因为嵌入式计算机和通用目的计算机之间最主要的差别在于它们的应用，两者的操作原则和内部基本体系结构都是相同的。

两者都有处理器、存储器以及一些输入输出形式。两者的首要差别在于用途的不同，而这点体现在它们所使用的软件上。在操作系统对系统资源的协调下，台式计算机可以使用各种应用程序。通过运行这些不同的应用程序，台式计算机的功能得以改变。有时它可以作为文字处理器，而有时它又可以作为一个MP3播放器或者数据库客户端，所有这些都是在用户的控制下通过加载和运行相应的软件来实现的。

与之相反，嵌入式计算机通常用于某种特殊任务。许多情况下，嵌入式系统用于取代专用电子设备，其优势在于，系统的功能是由软件而非硬件决定的。因此，比起复杂的电路，嵌入式系统更容易生产，也更容易发展。

典型的情况是嵌入式系统拥有一种且仅一种应用，该应用一直不停地运行着。嵌入式计算机可能有，也可能没有操作系统，并且它很少为用户提供随意安装新软件的能力。因为常用的软件已经包含在系统的非易失性存储器里，而不是像台式计算机那样，在非易失性存储器中只包含引导软件和（可能的）底层驱动。

嵌入式硬件通常较台式计算机简单得多，但也可以更为复杂。单个芯片加上少许支持部件就可以实现一个嵌入式计算机，而它的用途可能是如同一个花园灌溉系统控制器一样简单。这个嵌入式计算机也有可能是一个用于所有商业喷气式飞机的飞行控制系统，或拥有150个处理器的分布式并行机器。嵌入式硬件可能千差万别，但根本的设计原理都是一样的。

本章介绍了与计算机体系结构有关的一些重要概念，着重强调了那些与嵌入式系统有关的主题。其目的就是奠定读者阅读后续章节的背景知识。在本章里，你将学到的基本概念包括：处理器、中断（Interrupt）、RISC和CISC之间的差异，以及并行系统、内存和I/O系统。

概念

让我们从头说起。

简单来说，计算机是一台设计用来处理、存储和重新获取数据的机器。这些数据可以是电子表格里的数字，一个文档中文本的字符，图像中颜色的像素，声音的波形或者诸如空调或CD播放器这些系统的状态。注意到所有的这些数据在计算机中都是作为数字来存储的，这一点非常重要。在深入到C代码中时很容易忘记这一点，而是更关注于复杂的算法和数据结构。

计算机通过对这些数字执行操作来操纵这些数据。比如，计算机通过把一个数字阵列转移到显存中来完成一个图像在屏幕上的显示，每个数字代表一个颜色的像素；要播放

MP3 音频文件，计算机就从磁盘读一串数字到存储器中，对这些数字进行巧妙的处理，把压缩了的音频数据转换成原始音频数据，并将这些新的数字集（即原始音频数据）输出到音频芯片中。

计算机所做的任何事情，从网页浏览到打印，都包括对数字的转移和处理。计算机的电子设备只不过是设计用来保存、转移和改变这些数字的一个系统而已。

一个计算机系统是由既有软件又有硬件的诸多部件组成的。其核心部件是用来执行计算机程序的硬件设备：处理器。在一个系统中，计算机往往也有几种不同类型的存储器。存储器用来对处理器正在执行的程序进行保存，也对这些程序所操作的数据进行保存。计算机还有存储数据或者与外界交换数据的设备。这些外部设备允许通过键盘输入文本，在屏幕上显示信息，或者在磁盘驱动器之上移动程序和数据。

计算机软件控制着计算机的操作和功能。计算机中的软件可以分为许多所谓的“层”（如图 1-1 所示）。一般而言，一个给定的层只与直接相邻的上下层交互作用。

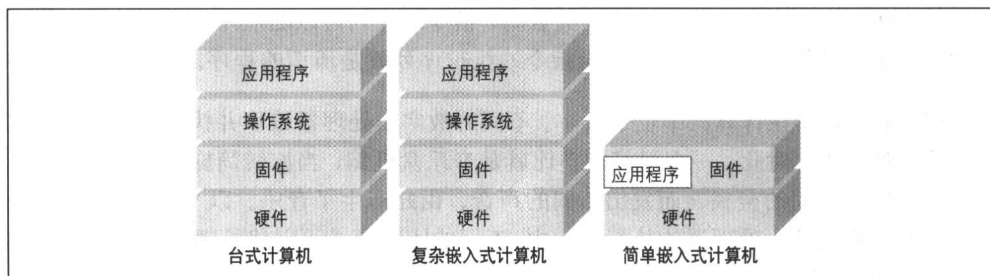


图 1-1：软件的层次

处在上述结构最底层的是在计算机刚启动时由处理器运行的程序。这些程序将其他硬件子系统初始化到一个已知的状态，并对其进行配置以保证计算机的正常运行。这种软件由于永久性地存储在计算机的存储器里而被称作固件（firmware）。

引导程序（bootloader）保存在固件中。引导程序是由处理器运行的一个特殊程序，它把操作系统从磁盘（或者非易失性存储器，或者网络）中读取到内存中，以便处理器随后运行该操作系统。在台式计算机和工作站中都有这种引导程序，在一些嵌入式计算机中也可能有。

在固件之上，由操作系统控制着计算机的操作。操作系统对存储器的使用进行组织，对诸如键盘、鼠标、显示器和磁盘驱动器之类的设备进行控制，等等。操作系统还是一种为用户提供接口以便用户运行应用程序、存取磁盘里的文件的系统软件。操作系统也为应用程序提供了一套软件工具，通过这种机制，这些应用程序也可以对显示器、磁盘驱

动器等这样的设备进行访问。并非所有的嵌入式计算机都使用乃至需要使用一个操作系统，嵌入式计算机常常运行面向特定任务的简单代码，这种情况下操作系统难免有大材小用之嫌。在另外一些场合，如网络路由器，操作系统提供了必要的软件集成，大大简化了开发过程。在嵌入式计算机中是否需要一个操作系统以及操作系统是否有用，实际取决于这一嵌入式计算机的预定用途，至少也取决于设计者的偏好。

在上述结构的最高层是应用软件（application software）。应用软件由为计算机提供功能模块的应用程序组成。应用软件层之下的所有东西都看作系统软件。对于嵌入式计算机而言，应用软件和系统软件的这种界限经常很模糊。这点也反映了嵌入式设计中的基本原则：系统应尽可能地以简单、直接的方式达到其目标。

处理器

处理器是计算机中最为重要的部件，所有的其他部件都是以此为中心。其实，处理器就是计算机的运算部件，是一个能够以顺序指令指定的方式对数据（信息）进行操作的电子器件。这里的指令也就是操作码或机器码，指令的序列可以随应用程序而改变。从这个角度上讲，计算机是可编程的，而指令的执行序列就是所谓的程序。

与数据一样，指令在计算机中也是数字。不同的数字由处理器读取并执行时，可以导致不同的事件发生。对此，一个很好的类比就是音乐盒机制，当旋转筒旋转时，不同的小突起依次碰击不同的金属簧而发出不同的声音，由此产生了音乐。以与此类似的方式，指令的比特模式流入处理器的执行单元中，不同的比特模式可以将处理核的不同部分激活或使之无效，这样，一个给定指令的比特模式可以激活一个加法操作，而另外一种比特模式可以使一个字节存储到存储器中。

指令的序列就是机器码程序。每一类型的处理器都有一个不同的指令集（instruction set），这意味着指令的功能（及激活它们的比特模式）是不同的。处理器的指令往往很简单，如“两个数相加”或“调用函数”等。然而，在有些处理器里，这些指令可能就复杂难懂了，如“如果前一操作结果为零，就用这一特殊数字来定位存储器里的另外一个数字，一旦完成后便递增第一个数字”。关于这点，将在稍后有关CISC和RISC处理器的章节里作详细介绍。

基本的系统体系结构

单个处理器是不能完成任何任务的，还需要存储器（用于程序和数据存储）、支持逻辑以及至少一个I/O设备（输入/输出设备），用于实现计算机与外界之间的数据传输。基本的计算机系统如图 1-2 所示。

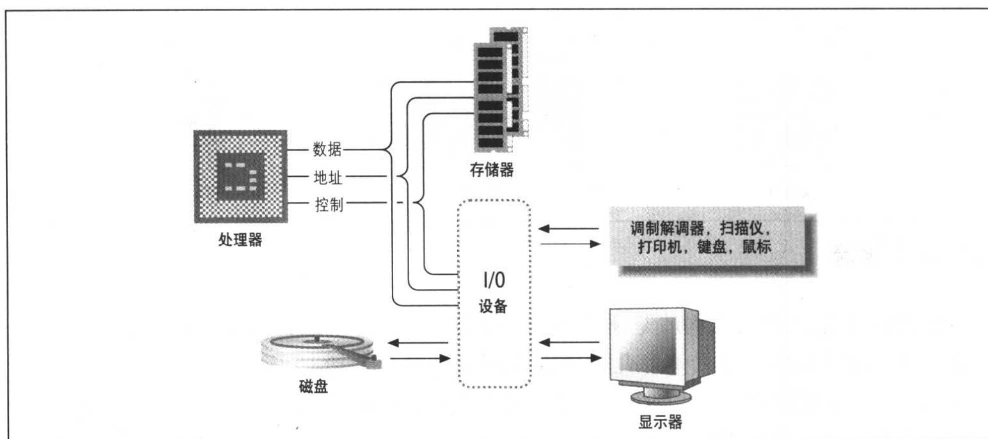


图 1-2：基本的计算机系统

微处理器（通常）是在一个单独的集成电路上实现的处理器。除了在一些大型超级计算机中存在差异之外，几乎所有的现代处理器都是微处理器，因此这两个术语经常交替使用。现在普遍使用的微处理器有 Intel Pentium 系列、Motorola/IBM PowerPC、MIPS、ARM 以及 Sun SPARC 等。微处理器有时也就指中央处理器（Central Processing Unit，CPU）。

微控制器在单一集成电路中包含了一个处理器、存储器以及一些 I/O 设备，可用于嵌入式系统。连接微处理器和 I/O 设备的总线也存在于同一集成电路中。现有的微控制器种类繁多，从小型 PIC 和 AVR（本书将谈到），到内部集成 I/O 功能的 PowerPC 微处理器，主要面向一些嵌入式应用。

微控制器与片上系统（System-on-Chip, SoC）很相似，后者主要针对诸如 PC 和工作站这样的一些常规计算机。SoC 处理器具有一套不同的输入/输出，反映其针对的应用，它对很多外部存储器提供了接口支持；而微控制器通常将它们的存储器集成到芯片上，并且只为外部存储设备提供有限的支持。

计算机系统的存储器包含了微处理器将要执行的指令及其将要操作的数据。计算机系统的存储器从来不会是空的，里面总是有指令、有用数据或者是系统启动时所产生的随机无用数据。

指令由计算机系统从存储器里读取，而数据则可以读出和写入存储器，如图 1-3 所示。

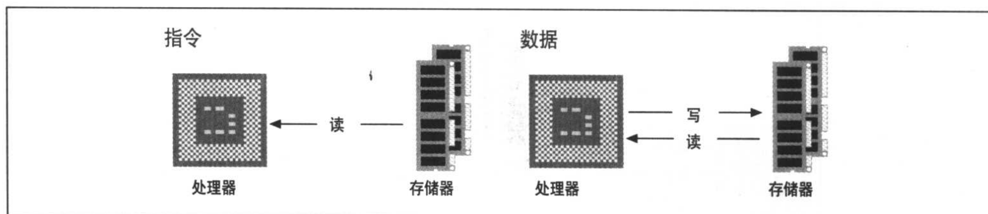


图 1-3：数据流

上述形式的计算机体系结构就是所谓的冯·诺伊曼机，它以这种体系结构的创始人之一约翰·冯·诺伊曼的名字命名。几乎所有的现代计算机都遵循冯·诺伊曼体系结构，很少有例外。冯·诺伊曼计算机又称为控制流计算机，在这种系统结构中，计算机的执行步骤是由程序的顺序控制所支配。换句话说，计算机会随着控制其运行的程序逐步执行。

注意： 还有一些有趣的非冯·诺伊曼体系结构，如可以进行大规模并行处理的 Connection Machine”（译注 1），以及尚在发展中的生物计算机和量子计算机或神经网络。

典型的冯·诺伊曼计算机具有以下几个明显特征：

在数据和指令之间没有本质差别。处理器能够从存储器中给定点处直接开始执行，而它本身却无法得知自该点开始执行的数字序列是指令还是数据。指令 0x4143 也可能是数据（数字形式的 0x4143 或者 ASCII 码的“A” 和“C”）。处理器也无法告知所执行的是数据还是指令。如果一个数字将由处理器执行，那么它就是指令；如果一个数字将由处理器操作，那么它就是数据。

由于缺乏对指令和数据的区别，处理器能够在程序控制下改变其指令（将指令作为数据来处理）的执行。也正是因为处理器无法识别数据和指令，它将盲目地执行所给的任何内容，而不管所执行的指令序列是否有意义。

数据没有固有含义。代表图像中颜色的一个像素的数字与代表文本文档中一个字符的数字，对处理器而言没有任何差别。这些数字的具体含义取决于在程序的执行。

数据和指令共享同一内存。这意味着一个程序的指令序列可以被另外一个程序当作数据来处理。计算机的编译器通过在存储器中生成数字（指令）序列来建立一个二进制

译注 1： Connection Machine（CM-1）是由 Danny Hillis 在 1986 年公开展示的第一台大规模并行处理超级计算机，给业界留下了不可磨灭的印象。这种计算机具有 64 000 个处理器，每个处理器都有自己的存储器。通过把问题分段化，并同时把不同的段分配给各个处理器执行，CM-1 能够处理复杂的计算和任务，而所耗费的时间却仅相当于一次只能处理一个问题的传统对称处理所需时间的一小部分。

程序，对于编译器而言，所编译的程序只是数据而已，因此它也就按照数据对其进行处理。只有当处理器开始执行这个二进制程序时，它才是一个程序。同样，操作系统通过把应用程序的指令序列看作是数据而从磁盘中加载到内存，进而实现上述数据/指令处理。程序如同图像或文本文件那样被加载到内存，这归功于共享内存空间。

内存是存储单元的线性（一维空间）排列。处理器的内存空间可能包含操作系统、各种各样的程序及其相关数据，所有这些都在同一线性空间之内。

内存空间内的每一存储单元都有一个唯一、有序的地址，内存单元的地址用来定位（和选择）该内存单元。内存空间也就是通常所说的地址空间，如何划分不同的内存与I/O设备之间的地址空间就是所谓的内存映射。地址空间就是所有可寻址存储单元的陈列，一个具有16位地址总线的8位处理器（如68Hcll），可以存取 2^{16} =65536=64K的内存。因此，我们会说，这个处理器具有64K的地址空间，一个具有32位地址总线的处理器则可以存取 2^{32} =4G的内存。

有些处理器，特别是Intel公司的x86系列，为I/O设备设置了单独的地址空间，并且有单独的指令来访问这样的空间，这就是I/O端口映射方式。然而，大多数处理器在地址空间之内并不对存储设备和I/O设备进行区分，I/O设备和存储设备存在于相同的线性空间内，使用相同的指令对这两种设备进行访问，这就是I/O内存映射方式（如图1-4所示）。I/O内存映射方式是目前最为常见的映射方式。I/O端口映射方式很少用了，这一术语也就用得更少了。

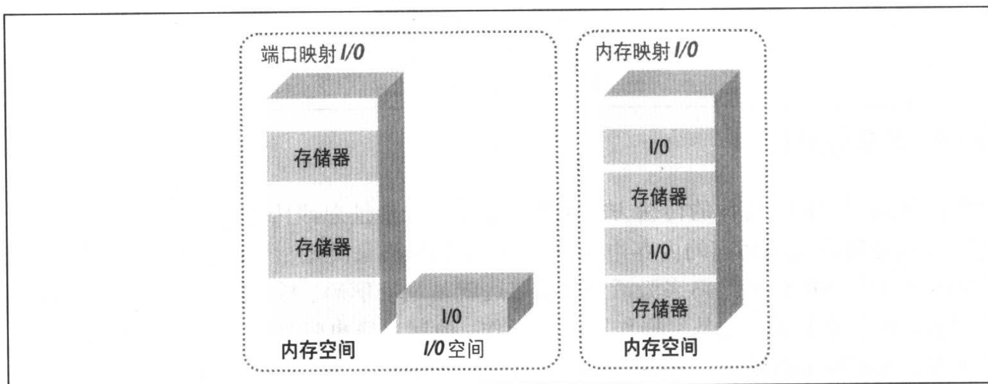


图1-4：I/O端口映射与内存映射对比

现有的大多数微处理器都是标准的冯·诺伊曼机。与这种体系结构相背的主要是哈佛体系结构（Harvard architecture），其指令和数据具有不同的内存空间（如图1-5所示），这样，每一内存空间里都有各自的地址、数据和控制总线。这种哈佛体系结构具有很多优

点，比如可以同时取指令和数据，以及指令字长并不由标准数据单元（word 字长）所决定。

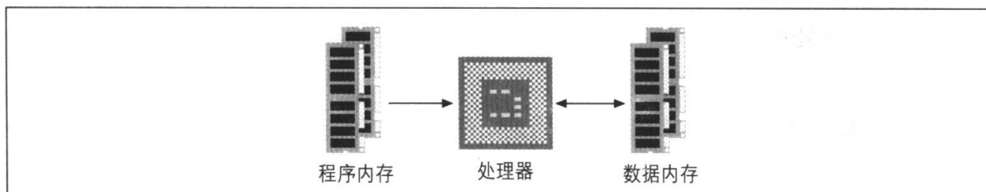


图 1-5：哈佛体系结构

总线

所谓总线，就是具有相关功能的信号线的物理集合。总线允许在计算机系统的不同部件之间传输电子信号，因此可把信息从一个设备传输到另外一个设备上。例如，数据总线是在处理器和组成计算机系统的不同子系统之间传输数据的信号线集合。总线的宽度是指用于传输信息的信号线的数目，如一个 8 位宽总线一次并行传输 8 位数据。

现有的大多数微处理器（也有一些例外）采用了三总线系统体系结构（如图 1-6 所示），分别是地址总线、数据总线和控制总线。

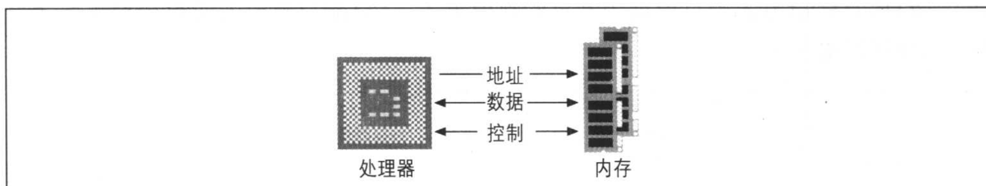


图 1-6：三总线系统

数据总线是双向的，具体的传输方向由处理器决定。地址总线传输的是地址信息，这一地址指向处理器试图访问的内存中的位置。由外部线路来决定指定的内存单元位于哪个外部设备上，并由外部线路来激活该设备。这就是地址解码。控制总线携带来自处理器的当前访问状态信息，如是写操作还是读操作。控制总线也将当前访问状态信息返回给处理器，例如地址错误等。不同的处理器具有不同的控制线，但许多处理器中有一些控制线是一样的。控制总线可能含有一些输出信号，如读、写、有效地址等。一个处理器也有几个输入控制线，如复位、一个或多个中断线以及时钟输入等。

注意： 几年前，我有幸驻足观察 CSIRAC（发音作“sigh-rack”）。CSIRAC 是在 20 世纪 40 年代末期设计并建造于澳大利亚悉尼的世界上最早的数字计算机之一。这是一台笨重的机器，填满了整个大大的屋子，有些坚固的部件你可以真的踢着玩。这真是一个回顾旧机器的一个经历。参观中，我记得其中有一阶段是步行穿越磁盘控制器（它有一个小房间那么大），仰望系于头顶的一大捆电线。我就问导游，那些电线是干吗的？他回答说：“那是数据总线！”

现在 CSIRAC 已经被收藏到澳大利亚墨尔本大学的博物馆里，你可以在线浏览这一机器，甚至可以从 *http://www.cs.mu.oz.au/csirac* 下载一个该机器的模拟器。

处理器操作

处理器能够实现六个基本的功能：向系统的内存写入数据或向 I/O 设备写入数据；从系统的内存读出数据或从 I/O 设备读出数据；从系统的内存读取指令；能够执行处理器里数据的内部操作。

在很多系统中，向内存中写入数据在功能上等同于向一个 I/O 设备中写入数据。同样地，从系统的内存中读取数据也就等同于从 I/O 设备中读取数据或者从内存中读取指令这样的外部操作。换句话说，处理器对待内存和 I/O 一视同仁。

处理器的内部数据存储由它的寄存器来完成。一个处理器拥有有限数量的寄存器，由这些寄存器来保存处理器正在操作的当前数据 / 操作数。

ALU

算术逻辑单元（Arithmetic Logic Unit, ALU）用于完成处理器里数据的内部算术操作。由处理器读取并执行的指令控制着寄存器和 ALU 之间的数据流向，进而经由 ALU 完成操作，通过 ALU 控制数据的输入。关于 ALU 的符号表示如图 1-7 所示。

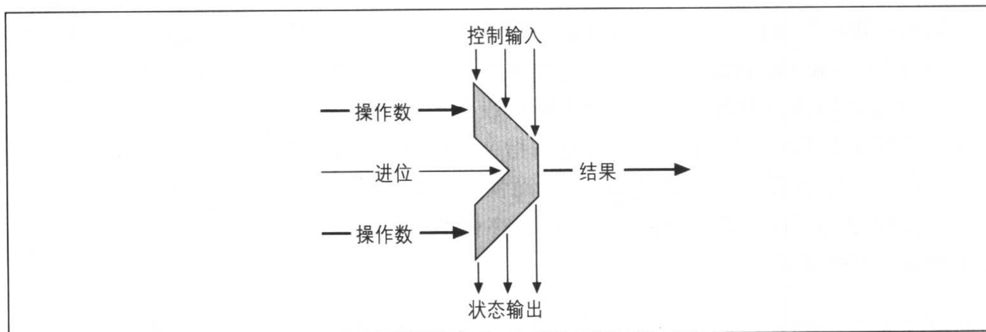


图 1-7：ALU 结构图

一旦收到处理器的指示，ALU 就对一个或多个值执行一个操作（典型的是下列运算之一：加、减、乘、除，逻辑运算NOT、AND、NAND、OR、NOR、XOR，移位操作算术左/右移位或循环左/右移位等）。ALU 所操作的值就是操作数，主要是从两个寄存器里或一个寄存器和一个内存单元中获得。ALU的运算结果被放置回目标寄存器或存储单元中。ALU 的状态输出表明有关一个 ALU 操作的所有特殊属性，如结构是否为零或为负，或者是否溢出或者产生进位等。一些处理器有单独的单元用于乘法、除法以及移位操作，提供了更快的操作和更高的吞吐量。

每一个计算机体系结构都有其自身的ALU特性，不同的处理器之间的ALU差异可能很大。然而，所有的这些差异只体现在某个方面，它们都具有上面所描述的共同特征。

中断

中断（在一些处理器里也叫作陷阱或异常）是一种转移处理器的当前程序执行点，以便处理器能够处理已发生的某些事件的技术。这里的事件可能是来自外部设备的一个错误，或者仅仅是 I/O 设备已经完成上一个 I/O 任务而等待另外一个新的任务。当你在计算机上敲击键盘或晃动鼠标时，就会产生一个中断。可以把中断看作一个硬件产生的功能调用。

中断技术大大减轻了处理器不断检查I/O设备是否需要服务的负荷，使得处理器可以继续进行其他任务。当I/O设备需要处理器的介入时，它就会发出一个中断输入来通知处理器。在一些处理器中，中断可被赋予不同的优先级，因此对能够中断处理器的不同事件指派不同的重要性。正服务于低优先级中断的处理器可以暂停当前任务而为高优先级中断服务。当然，如果此时有更低优先级的中断产生，处理器将忽略此中断直至它完成当前相对较高优先级的中断服务。

当一个中断产生时，处理器就通过把寄存器和程序计数器等压入堆栈来保存当前执行状态。接着，处理器就把相应的中断向量加载到程序计数器。中断向量就是中断服务例程（Interrupt Service Routine, ISR）所在的地址。这样，加载中断向量到程序计数器也就导致处理器开始执行ISR，完成产生中断的设备的任何服务请求。ISR的最后一条指令通常是中断返回指令。通过这一中断返回指令，处理器能够从堆栈中重装它所保存的前一程序状态（寄存器及程序计数器），从而恢复原有的程序执行。中断在很大程度上对原先的程序是透明的，这就是说，原先的程序根本“没有察觉”处理器已经接受中断了，除了损失一些中断时间间隔之外。

带有影子寄存器的处理器使用这些影子寄存器来保存程序当前的状态，而不是通过将寄存器压入堆栈。这就使得处理一个中断时节约了可观的访存时间（也就节省了中断处理时间）。然而，因为处理器中只有一套影子寄存器，因此在服务更高级的中断请求之前，

服务于多中断的处理器必须"手工地"保存当前寄存器的状态。否则，重要的状态信息就会丢失。在中断从ISR返回时，影子寄存器里的内容就交换到主要寄存器组中。

硬件中断

有两种方式可以判断何时一个I/O设备（如串行控制器或磁盘控制器）已准备好了下一次将要传输的数据序列。第一种是忙等待或轮询检测，此时处理器不停地检查设备的状态寄存器直到设备就绪。这种方式相当浪费处理器的时间，但实现起来最为简单。

一个更好的方式是当设备准备好一个传输发生时，就对处理器产生一个中断。小型、简单处理器可能仅有一个（或两个）中断输入，因此数个外部设备就不得不共享处理器的中断线。当有中断产生时，处理器就必须检查每一个设备以确定哪一个设备产生了中断（这也可以认为是轮询的一种形式）。中断轮询较普通轮询的优势在于，轮询操作仅当处理器有必要服务于一个外部设备时才发生。中断轮询方式仅适用于具有少量外部设备的系统，否则，处理器将花费大量的时间来确定中断源。

服务中断的另外一个技术是使用向量中断（注1），通过向量中断，中断设备能够确定处理器将要执行哪一个中断向量。采用向量中断技术可以极大地降低处理器花费在确定中断源上的时间。如果一个中断请求由多个中断源产生，那么就有必要对不同的中断源分配不同的优先级。这可以通过软件或硬件来解决，具体采用哪种方式依赖于特定应用。在这种方案中，处理器具有多根中断线，每一个中断对应于一个给定的中断向量。因此，举例来讲，当一个中断优先级为7的中断发生时（相应于7的中断线被宣布占用），处理器就把向量7加载到程序计数器里，并开始执行由中断7所确定的服务例程。

向量中断可以更进一步。当某些处理器和外部设备支持当设备产生中断时真正由设备把适当的向量放到数据总线上，这就意味着系统更为多功能了，所以并非每个外部设备局限于一个中断向量，每一个设备能够为引起中断的事件提供一个指定的中断向量。然而，要实现这种方案的话，处理器必须支持这种特征，但实际上大多数处理器并不支持。

有些处理器具有一种所谓的快速硬件中断的特征。这种中断方式只需保存程序计数器即可，它假定通过手工地保存寄存器的内容来使中断服务例程保护必需的寄存器的状态。当一个I/O设备需要来自处理器的快速响应，而非等待处理器把所有的寄存器值保存到堆栈中时，快速中断方式就显得非常有用了。在这些处理器里，一个特殊的（且分离的）中断线用于产生一个快速中断。

注1：　注意，请区别于内存中的中断向量（Interrupt Vector）。

软件中断

软件中断是由指令产生的中断，具有最低的优先级，通常由程序用此来请求一个由系统软件（操作系统或固件）为其完成的服务。

那为什么还使用软件中断呢？为什么不直接调用适当的代码段？既然如此，为什么从根本上我们要使用操作系统来完成任务？这就回到了计算机兼容性问题上来了。程序跳转到子例程（调用函数）就是跳转到一个指定内存地址，而未来版本的系统软件可能不会把子例程的地址定位在旧版本系统软件所定位的同一地址上。通过使用软件中断，我们的程序就不必知道例程的具体位置，因为例程的正确位置依赖于向量表里的条目来指向。

CISC 和 RISC 处理器

处理器系统结构主要有两大分支：其一是 CISC（Complex Instruction Set Computer，复杂指令集计算机），其二是 RISC（Reduced Instruction Set Computer，精简指令集计算机）。标准的CISC处理器有Intel 公司的x86系列、Motorola的68xxx系列以及National Semiconductor 的 32xxx 系列，甚至于在某种程度上，Intel 公司的 Pentium 系列也属于 CISC 体系结构。普通 RISC 体系结构的处理器有 Motorola/IBM 的 PowerPC、MIPS 体系结构，Sun 公司的 SPARC、ARM，Atmel AVR 以及 Microchip PIC 等。

CISC处理器有一个单一处理单元、外部存储器、一个相对较小的寄存器集以及几百条不同的指令。在许多方面，CISC 处理器只不过是 20 世纪 60 年代大型机的小型版。

从20世纪70年代末到80年代初整个时期，处理器的设计就朝着越来越大而且越来越复杂的指令集的方向发展。需要从一个I/O端口传输一个字符串么？很好，在CISC处理器（80x86 系列）里，只需一条指令就能完成上述任务！在一些处理器里，CISC 处理器指令多种多样，很容易就超过上千条操作码，如Motorola的68000处理器。这种方式的优势在于它使汇编语言程序员编程更为容易 —— 程序员只需编写少许代码就可以完成任务。由于存储器曾经是缓慢和昂贵的，这种方式对于让每一条指令完成更多的工作来讲具有一定的意义，因为它减少了完成给定功能所需的指令数，进而降低了存储空间以及读取所需指令的访存次数。随着内存价格的不断变低及速度的不断提高，加上编译器的效率越来越高，CISC 方式的处理器的相对优势也就逐渐消失了。CISC 处理器的主要缺点之一就是这种处理器的本身变得越来越复杂了，这是由于处理器要支持这样一个庞大而且多样的指令集，使得控制和指令解码单元变得复杂且迟缓起来，硅片越来越大且难以生产，并且这些功能部件不但耗电而且导致了大量的热量产生。随着处理器越来越先进，CISC 在硅片上的开销就变得难以忍受了。

当单独考虑一个特定处理器的特性时，可能会增加处理器的性能，但是，如果这一特性增加了整个设备的总体复杂性，那么它实际很可能会降低系统的整体性能。人们发现，

通过将指令集简化为最普遍使用的指令，处理器就变得简单快速起来。通过这种方式，解码并执行每一指令只需很少的指令周期，这样执行一个程序所需的总的指令周期就大大缩短了。其缺点就是完成一个任务需要更多的（更简单的）指令，但此缺点在处理器性能的提升上得到了弥补。例如，如果时间周期和每条指令的周期数都降低为原来的四分之一，而完成任务所需的指令数增加了 50%，那么处理器的执行就提高了 8 倍。

这种实现导致了对处理器设计的再思考。其结果就是 RISC 体系结构，它导致了更高性能处理器的发展。RISC 背后的基本原则是，把硅片的复杂性转移到语言编译器里，硬件部分尽可能地保持简单和快速。

一个给定的复杂指令可以通过一系列简单的指令来实现。举例来说，许多处理器都有一个针对位操作的异或（xor，eXclusive OR）指令，也有一个清空指令（clear）来给一个指定的寄存器清零。然而，给一个寄存器清零也可以通过使用 XOR 指令本身来实现。这样，就不再需要单独的寄存器 clear 指令了，因为现有的 XOR 指令可以代替它。此外，有些处理器能够通过直接向内存单元写入零来将其清空，同样的功能也可以通过先把寄存器清零然后把该寄存器内容写入到这一内存单元来实现。向一个寄存器加载一个数值的指令可以由一个寄存器清空指令来代替，该清空指令后跟着一条以指令所带数值的为操作数的加法指令。这样，六条指令（xor、clear reg、clear memory、load literal、store 和 add）所完成的功能就可以只用三条指令（xor、store 和 add）来完成。

因此，下列 CISC 汇编伪代码：

```
clear 0x1000     ; 清空内存单元 0x1000
load  r1,#5      ; 向寄存器 1 加载值 5
```

就变成 RISC 汇编伪代码：

```
xor   r1,r1      ; 清空寄存器 1
store r1,0x1000  ; 清空内存单元 0x1000
add   r1,#5      ; 向寄存器 1 加载值 5
```

这将导致代码规模加大，但通过降低指令解码单元的复杂性就能加快整个操作。为了实现 RISC 的简单性，程序中存在许多这样的代码优化。

RISC处理器具有许多显著的特性，它具有大量的寄存器组（在一些体系结构里超过一千个），从而减少了处理器访问主存储器的次数。经常使用的变量就可以保留在处理器内部，这就减少了处理器访问外部存储器（比较慢）的次数。高级语言编译器（如 C 语言编译器）就利用了这点来优化处理器的性能。

通过采用更小和更简单的指令解码单元，RISC处理器具有很高的指令执行速度，同时也降低了处理单元的尺寸和功耗。一般而言，执行一条 RISC 指令只用一到两个时钟周期

(这极大地依赖于特定的处理器)。这就与CISC处理器形成了对比，在CISC中执行一条指令可能花费几十个周期。例如，一条指令（整数乘法）在一个80486 CISC处理器上要花费42个周期才能完成，而同样的指令在一个RISC处理器上可能只需一个周期。RISC处理器的指令格式很简单，所有的指令基本上具有同样的长度(这就使得指令解码单元比较简单)。

RISC处理器实现了所谓的Load/Store体系结构，这意味着真正访问内存的指令就是Load和Store。与此相反，许多（大多数）CISC处理器的指令能够访问或操作内存。在RISC处理器里，所有其他的指令（除了Load和Store）都只工作在寄存器上。这点推动了RISC处理器单周期内完成（绝大部分）指令的特征。因而，RISC处理器并没有CISC处理器中的所有寻址模式。

RISC处理器也经常采用流水线方式执行指令。这意味着，当一条指令正在执行时，也正在对指令序列中的下一条指令进行解码，与此同时第三条指令也在取指过程当中。在任何一个给定的时刻，有几条指令将在流水线中处于执行过程中。这又提高了处理器的性能。如此一来，即便并非所有的指令都在一个周期内完成，处理器也能够在每一周期上都引入和退出指令，据此实现有效的单周期执行。有些RISC处理器重迭执行指令，Load操作可以允许在所加载的数据还没有从内存中返回之前，指令序列中与之不相关的指令继续执行。这就允许这些指令与Load指令重迭执行，从而提高了处理器性能。

由于其计算能力和低功耗的特点，RISC处理器得到了广泛的应用，尤其是在嵌入式计算机系统中，并且一些RISC体系结构所具有的属性在传统意义上的CISC体系结构（如Intel公司的Pentium处理器）中也出现了。具有讽刺意味的是，许多RISC体系结构中增添了一些类CISC的特征，因此RISC和CISC之间的区别变得模糊起来。

有关RISC体系结构和处理器性能主题的精彩讨论可以从Kevin Dowd和Charles Severance合著的《High Performance Computing》一书中得到，该书已由O'Reilly出版。

这样看来，究竟是RISC还是CISC更适合于嵌入式和工业应用呢？如果功耗要求比较低，那么RISC体系结构可能是比较好的选择；如果程序的可用存储空间比较小，那么CISC处理器也许是很好的选择，因为在这里CISC指令对字节进行了更多的“锤炼”。

DSP

DSP（Digital Signal Processor，数字信号处理器）体系结构是另一种特殊类型的处理器体系结构，这些处理器具有优化处理阵列数据的指令集和体系结构。这种处理器通常扩展了哈佛体系结构的概念，不只是分为数据空间和代码空间，而且在数据空间内又为两个或多个块。这一方式允许为多操作数进行并行取指和数据访问。因此，DSP可具有很高的吞吐量，能够在某些应用中胜过CISC和RISC处理器。

DSP中有专用的硬件适合于矩阵数据处理，它们常常有硬件循环，藉此，专用寄存器容许并控制着指令序列的循环执行。这种硬件循环也就是通常所说的零开销循环，因为没有明确需要软件检查的条件来作为循环处理的一部分。DSP中也有专门的硬件来加速算术操作。高速乘法器、乘法累加器（Multiply-and-Accumulate，MAC）以及桶形移位器都是DSP的共同特性。

DSP一般用于嵌入式应用中，许多传统的嵌入式微控制器都含有一些DSP功能块。

存储器

存储器用来为处理器存放数据和软件，其类型多样，经常在单个系统内混合了各种各样的存储器类型。有些存储器能够在断电情况下保持里面的内容，但访问起来比较慢；有些存储器容量很大，但需要额外的支持电路，访问起来更慢；更有一些存储器设备拿容量来换取速度，这种存储器相对较小，但能够跟上最高速的处理器。

存储器芯片可以由两种方式来组织，可以是字组织（word-organized）或是位组织（bit-organized）。在字组织方案中，全部的半字节、字节或字都存储在一个元件里；而在位组织方式的存储器里，一个字节或者字的每一个位都被分配到一个单独的元件里（如图1-8所示）。

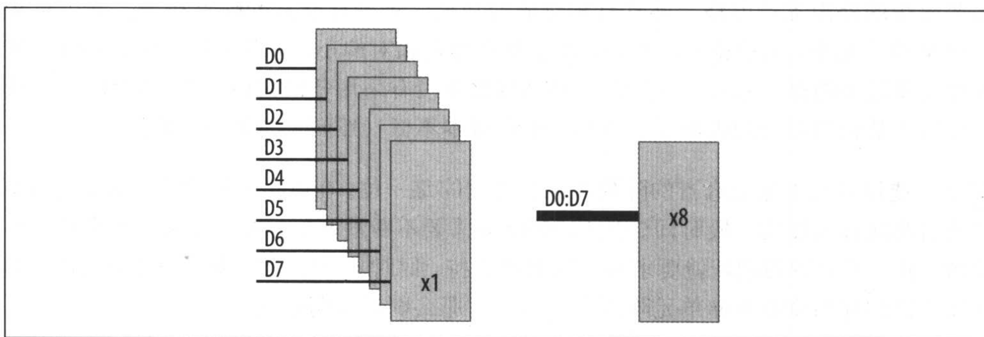


图1-8：8个位组织8×1位存储器设备和一个字组织1×8存储器设备

存储器芯片具有不同的大小，存储器的位宽用来描述其大小的一部分。例如，一个DRAM（动态RAM）芯片可以描述成4M×1（位组织），而一个SRAM（静态RAM）可以描述为512K×8（字组织）。在上述两种存储器里，每一个芯片拥有完全相同的容量，但组织方式却不相同。在DRAM里，为了匹配一个8位数据总线，需要8个芯片来组成一个存储器块，而SRAM只需一个芯片就够了。

然而，由于 DRAM 是以并行方式组织的，因而可以被同时访问。DRAM 存储块的最终大小就是（4M×1 位）×8 个设备，即 32M。在一个存储器模块上放置多块 DRAM 是比较普遍的做法，这也是 DRAM 安装在标准计算机上的通用做法。

存储器芯片常见的位宽有 x1、x4 和 x8 几种，尽管 x16 位宽的存储器件也有了。一个 32 位宽的总线可以有以下实现方式：32×1、8×4 或 4×8。

RAM

RAM 代表随机访问存储器（Random Access Memory），这个叫法其实有点不恰当，因为大多数（所有）计算机存储器都可以看作是“随机访问”的。RAM 是计算机系统中“工作着的存储器”，处理器可以容易地将数据写入 RAM 作为临时存储。RAM 一般是易失的，当系统断电时里面的内容就会丢失，任何必须保留的存于 RAM 的信息，在系统断电之前必须写入到某些永久性的存储部件里。也有一些特殊的非易失性 RAM，里面集成了一个电源备份系统，因此甚至在计算机系统关机时 RAM 里面仍可以保留供电。

RAM 通常分为两类：静态 RAM（static RAM，SRAM）和动态 RAM（dynamic RAM，DRAM）。

SRAM 使用一对逻辑门来保存数据的每一位，是 RAM 里面最为快速的一种存储器，仅需要很少的外围支持电路，并具有相对低的功耗。这种存储器的缺点在于，其容量较 DRAM 明显要小，但价格却高很多。由于单个容量相对偏小，所以完成同样的存储容量就得需要较多的存储芯片。一台除了 SRAM 之外没有采用任何其他存储器的现代 PC 就可以看作是相当强大的机器了，当然，生产成本不低（然而，它的确非常快）。

DRAM 使用本质上是电容器的阵列来保存数据的每一位。在电荷开始消失之前，电容器仅能将其保存很短的一段时间。因此，DRAM 需要不断地刷新，大概每隔几个毫秒就得刷新一次。不间断的刷新就要求额外的电路支持，同时也会延迟处理器访存的时间。如果处理器访存与刷新存储单元阵列产生冲突，那么刷新周期必须优先。

DRAM 是现有的最高容量的存储器器件，具有广泛多样的衍生器件。通过接口把 DRAM 连接到小的微控制器上一般不可能，当然也不现实。大多数支持大地址空间的处理器都包含对 DRAM 的支持。将 DRAM 连接到这样的处理器上只是对引脚（或插脚）进行简单的连接而已。对于不支持 DRAM 的处理器，通过一个特殊的 DRAM 控制芯片来连接 DRAM 实际上也非常简单。

许多处理器有指令和/或数据高速缓存，里面存放着存储器最近访问的数据。这些高速缓存常常位于处理器内部，通常是用高速存储单元和高速数据通道来实现的。执行的指令通常来自指令高速缓存，因此提供了高速指令执行。如果一个访问高速缓存没有命中，

处理器能够快速地从主存中重新加载高速缓存。有些处理器具有逻辑电路，通过这一逻辑电路能够预测高速缓存没有命中，进而在未命中发生之前先加载高速缓存。高速缓存是用快速SRAM来实现的，在一个大系统里通常用来弥补主存DRAM的缓慢。

ROM

ROM代表只读存储器（Read-Only Memory）。这个叫法也有点不恰当，因为好多（现代）ROM也能够写入。ROM是非易失性存储器，不需要电流来维持所存数据。这种存储器通常比RAM慢，当然比SRAM更慢了。

系统里的ROM的首要用途就是保存系统启动时需要用到的代码（有时是数据）。这种软件通常也叫固件，包含了通过把I/O设备置于可知状态来初始化计算机的软件；固件里面也可能包含一个引导装入程序来从磁盘或网络上加载一个操作系统，如果是在一个嵌入式系统里，那么也可能包含应用软件本身。

许多微控制器里含有片上ROM，这就减少了部件数目，简化了系统设计。

标准ROM是由大量二极管阵列装配的（从某种简单化的意义上讲）。ROM中未写入的位状态都是1，每一个字节单元都读作0xFF。把软件加载到ROM中的过程也就是烧录ROM。这一术语来自于如下事实：编程过程是通过向适当的二极管注入一个足够大的电流来将其“烧断”或者烧录来完成的，从而在那一存储单元创建了一个零状态。一个叫作ROM烧录器的设备能够完成上述过程；如果系统支持，ROM也可能在电路里编程实现。这种方式叫作芯片集成系统编程（In-System Programming，ISP），或者有时叫作在线编程（In-Circuit Programming，ICP）。

一次性可编程（One-Time Programmable，OTP）ROM，顾名思义，这种ROM只能烧录一次。如果固件已经稳定并且产品要批量向客户提供，那么计算机制造商在系统中一般都使用这种ROM。掩膜可编程ROM也是一次性烧录的，与OTP ROM不同，它们是由芯片制造商在出售之前烧录的。与OTP ROM类似的是，一旦所烧录软件被认为是稳定的，就使用掩膜可编程ROM，并且大规模出货具有降低生产成本的优点。

EPROM

在最终产品中，一次性可编程ROM出货量是庞大的，但在调试时，这种ROM却是非常浪费的，因为所烧录软件的每一次反复，就要烧录一个新的芯片并丢弃原先的芯片。这样，一次性可编程ROM对开发而言就非常高了。

对系统开发和调试来说，一个更好的选择是可擦写ROM，即EPROM（Erasable Programmable Read-Only Memory）。通过对芯片上部的一个窗口照射紫外线就可以擦

除里面的数据，以此实现EPROM的再编程和复用。这种芯片的引脚和信号与一次性可编程ROM和掩膜ROM兼容。因此，研发时就可以使用EPROM，在生产时则可以使用一次性可编程ROM，而对系统的其他部分没有任何影响。

EPROM与其孪生姊妹OTP ROM的容量从几千字节（如今已经很少见了）到几兆字节甚至更多。

EPROM技术的缺点在于，如果要对其重写，就必须把芯片从电路上取下来，并且完成擦除也要好几分钟。擦除完毕，还要将其放到烧录器上，加载程序，然后再放回电路上。这个过程导致了很慢的调试周期。此外，对于存储可变系统参数而言，这种存储器显然没有用处。

EEROM

EEROM是电可擦除只读存储器（Electrically Erasable Read-Only Memory），也叫作电擦除可编程只读存储器（Electrically Erasable Programmable Read-Only Memory，EEPROM），但很少叫作电可改写只读存储器（Electrically Alterable Read-Only Memory，EAROM）。EEROM的发音是“e-e ROM”或“e-squared ROM”，或者有时候简称为“e-squared”。

EEROM可以在电路上进行擦除和再编程。它们的容量较标准的ROM（通常也只有几千字节）明显小得多，因此这类存储器不适用于保存固件。它们一般用于断电期间保存系统参数和模式信息。

许多微控制器通常把一个很小的EEROM合并到芯片上来保存系统参数。在嵌入式系统里，这就非常有用了，可以用来存储网络地址、配置信息、序列号、服务记录等等。

闪存

闪存（Flash）是最新的ROM技术，并且快速地占据了优势地位。闪存具有EEROM可再编程的特点，也具有标准ROM大容量的优势。闪存芯片有时候当作“Flash ROM”或“Flash RAM”，因为它既不像标准的ROM，也不像标准的RAM，我更喜欢将其直接叫作闪存以免混淆。

闪存通常按照扇区来组织，其优点在于可以擦除重写单个扇区而不影响设备的其他部分里的内容。它的特点是在写入一个扇区之前必须先将其擦除，而不能像RAM那样写覆盖。

有几种不同的闪存技术，不同的芯片制造商之间其闪存设备的擦除和编程需求都不同。

输入/输出

除了内存地址以外，处理器的地址空间也可以包含设备地址。这些设备就是I/O设备（也叫外部设备），处理器通过这些设备与外部世界进行通信。这类设备包括，与键盘、鼠标、modem等设备进行通信的串口控制器，以及控制一些外部子系统的并行I/O设备、磁盘控制器、视频控制器或者网络接口等。

处理器与外部世界交换数据的方式主要有三种：

程控输入输出（Programmed I/O, PIO）

处理器不时地在其方便的时候接收或发送数据。

中断驱动I/O （Interrupt-driven I/O）

外部事件通过向处理器请求当前运行程序挂起和执行外部事件服务而控制处理器。当一个外部设备中断处理器（向处理器宣布占用一个中断控制线）时，处理器就挂起当前任务（程序），开始执行一个中断服务例程。中断服务可能包含从输入设备向内存或从内存向输出设备传输数据。

直接存储器访问（Direct Memory Access, DMA）

允许在没有处理器不断参与的情况下在主存和I/O设备之间直接传输数据。DMA一般用在数据传输率非常重要的高速系统中。并不是所有的处理器都支持DMA操作。

DMA

DMA是在内存的两个区域之间或者在内存和I/O设备之间以流水线方式传输大块数据的一种方式。假设你要从硬盘读取100MB数据，并将其存到内存中，那么你有两种方式可以选择。

第一种方式是，处理器每次从磁盘控制器读取一个字节到寄存器里，然后将寄存器的内容保存到适当的内存单元。在这种方式中，每传输一个字节，处理器都必须读取一条指令，解码该指令，读取数据，读取下一条指令，解码下一条指令，然后存储所读数据。接着处理器开始传输下一个字节，如此反复直至读完所有数据。

第二种方式是，采用DMA在系统间传输大块数据。用一个叫作DMA控制器（DMA Controller, DMAC）的设备来完成内存和I/O设备之间的高速数据传输。这种数据传输方式通过在I/O设备和内存之间建立一条通道而绕开处理器。这样，数据从I/O设备里读出，再写到内存中去，这个过程中无需执行代码来逐字节或逐字地传输数据。

为了进行DMA传输，DMA控制器必须获得地址和数据总线的使用权。系统设计者可以选择几种方式来实现此目标。最为常用（也可能是最简单）的方式是挂起处理器的操作，让处理器释放总线（总线是三态的），这样DMAC便可以为完成数据传输而在短时间内获取总线使用权。支持DMA操作的处理器通常有一个特殊的控制输入来使DMAC（或其他一些处理器）能够请求总线（译注2）。

有四种基本类型的DMA操作：

标准块传输

这种方式由完成一系列内存传输的DMA控制器来实现，传输中包含一个从源地址中的加载操作（load operation），之后是一个到目标地址的保存操作（store operation）。标准块传输方式是在软件控制下初始化的，用来将数据结构从内存中的一块区域移动到另外一块区域。

命令模式传输

这种方式很像标准块传输方式，只是传输操作是由外部设备来控制的。命令模式传输用于内存和I/O设备之间的相互传输数据，由I/O设备来请求和同步数据的移动。

连续传输

这种方式提供了系统内的高速数据传输。不同于常规的多总线访问DMA传输方式，连续传输方式通过一次访问就能在数据源和目的地之间进行传输。数据在没有到达目的地之前是不会被读入处理器的。在连续传输方式中，内存和I/O设备分别获取不同的控制信号。例如，在内存收到写请求信号的同时，I/O设备也得到了一个读请求信号。数据从I/O设备直接转移到内存设备。

数据链传输（Data-chaining Transfers）

数据链传输方式允许按照内存中的链表的指定来完成DMA传输。这种方式开始于指向内存中一个描述符的指针。这个描述符是一张表，表里描述了所要传输的字节数、源地址、目的地址以及指向下一个描述符的指针。DMA控制器从这个表里加载与传输相关的信息后就开始传输数据。这一传输持续进行，直到已传输的字节数等于描述符里字节数域的数值。一旦完成本次传输，就加载指向下一个描述符的指针。数据传输操作继续进行直至发现空指针。

为了说明DMA的使用，我们来考虑一个从磁盘控制器到内存采用连续传输方式进行DMA操作的例子。一个DMA传输从处理器为传输配置DMAC开始，这个步骤包括指定数据的源地址、目的地址、大小以及其他一些参数。磁盘控制器向DMAC（而不是处理器）发送一个服务请求；DMAC就向处理器发送一个HOLD或BR（Bus Request）请

译注2：CPU一般在执行完当前指令的当前的总线周期后，向DMAC发出总线响应信号。

求；处理器执行完当前指令，将地址、控制和数据总线置于高阻状态（漂移、三态或者释放它们），然后向DMAC响应一个HOLD确认或BG（Bus Granted）信息，接着进入休眠状态。在收到HOLD确认信息之后，磁盘控制器到内存的传输就开始了，DMAC把内存单元地址放到地址总线上，并发出写内存的信号，与此同时，磁盘控制器把数据放到数据总线上。至此，从磁盘控制器到内存的一个直接访存就完成了。

以同样的方式，从内存到I/O设备的数据传输也可能实现。DMAC能够胜任处理数据的块传输。当I/O设备产生（或收到）数据时，DMAC就自动增加地址总线上的地址使其指向每一个连续的内存单元。一旦数据传输完毕，总线就归还给处理器而恢复其正常操作。

并非所有的DMA控制器都支持DMA的所有这些操作方式。有些DMA控制器只是简单地从源地址中读取数据，保存在内部，然后把数据存到目的地。它们几乎以与处理器同样的方式进行数据传输。这种DMA控制器传输较处理器传输的优势在于，后者的每一次传输都伴随着程序取指。这样，即便是进行由DMAC控制的顺序读和写传输，DMA控制器也不必取指和执行指令，从而提供了很高的数据传输率。

一些小型微控制器并不支持DMA操作。有些中型处理器（16位和低端32位的）可能支持DMA操作。所有的高端处理器（32位及其以上的）都支持DMA，并且许多带有片上DMA控制器。同样地，为小型计算机而设的外部设备也不提供DMA支持，而高速和强大的计算机的外部设备肯定支持DMA操作。

并行和分布式计算机

单处理器的性能已不能满足某些嵌入式应用的需要。由于价格的原因，实现一个带有最新超标量RISC处理器的设计可能很不现实，或者这个应用本身更适合于由几个相互通信的机器来分布处理。使用一系列分散安装的低价处理器可能会更有意义。使用并行处理器来实现嵌入式系统变得日益普遍。

并行体系结构介绍

传统的计算机体系结构遵循常规的冯·诺伊曼体系结构。基于这种体系结构的计算机通常只有单一、有序的处理器。这种计算体系结构的主要局限性在于，常规处理器只能一次执行一条指令。因此，运行在这些机器上的算法必须表达为一个有序问题。一个给定的任务必须被拆分为一系列有序的步骤，每一步骤都按序执行，且每次仅执行一个。

许多计算密集型的问题也是高并行性的问题。一个运用于大型数据集的算法就表现出这些问题的特征。对数据集里的元素的计算常常是一样的，并且这种计算松散地依赖于相

邻数据的计算结果。这样，以并行的方式对数据集里的每一元素完成计算的话，就可以获取很大的速度优势，远胜于从数据集里顺序移动数据并以串行的方式计算每一结果的计算方式。在这类应用中，能以并行的方式处理一个数据结构的多处理器机器就胜过常规计算机。

计算机的粒度由机器内正在处理的元件的个数决定。一台粗粒度机器含有相对较少的处理器，而一台细粒度机器就可能有好几万个处理元件。典型的情况是，一台细粒度机器处理元件的处理能力较粗粒度计算机里的处理元件的功能小得多。处理机的处理能力是通过大量的处理元件的强力方式获得的。

有多种不同形式的并行机器，每一种体系结构都有其自身的优势和局限性，并且每一体系结构均有一些支持者。

SIMD 计算机

SIMD（Single-Instruction Multiple-Data，单指令流多数据流）计算机是高并行性机器，采用了简单处理元件的大规模阵列。在一个 SIMD 机器里，每一处理元件均有一些局部存储器。SIMD 计算机所执行的指令从一个中央指令服务器广播到机器内的每一台处理元件。由于每一个处理器在其本地数据上执行指令，则数据结构里的所有元件就会得到机器的同时处理。

SIMD 机器通常与常规计算机协作使用。由 Thinking Machines 公司生产的 CM-1（Connection Machine）机器就是这种情况，这台机器不是使用 VAX 小型机，就是使用 Silicon Graphics 或 Sun 公司的工作站来作为“主机”。CM-1 是一台细粒度 SIMD 计算机，拥有总共可达 64K 的处理元件，这对主机系统来说就如同一块 64K 的“智能存储器”。运行在主机系统上的应用软件把数据集下载到 CM-1 上的处理器阵列，CM-1 里的每一个处理器作为单一存储器来运行。主机系统向 CM-1 的每一个处理元件同时传送指令，在计算完成后，主机系统从 CM-1 里读回运算结果，就如同从常规存储器里读取那样。

SIMD 机器的首要优势在于，通过简单和廉价的处理元件就能组建一台计算机。这样，通过廉价、现成的部件就可以获得功能强大的计算能力。并且，由于每一个处理器执行同样的指令，因此共享一个通用指令取指，因此这种机器的体系结构更加简单。对整台机器而言，只需存储一条指令即可。

使用多处理元件，每一个处理元件协调地执行同样的指令，这也是 SIMD 的主要缺点。有许多问题并不适于拆分为合乎 SIMD 计算机执行的形式。另外，一个给定问题的数据集很可能不怎么匹配给定的 SIMD 体系结构。例如，一个带有 10K 处理元件的 SIMD 机器就不能有效地配合一个拥有 12K 数据元素的数据集。

MIMD 计算机

另外一种主要形式的并行机是MIMD（Mutiple-Instruction Multiple-Data，多指令流多数据流）计算机。这种机器的特点是半自治处理器的粗粒度集合，每一个处理器都有自己的局部存储器和本地程序。执行于一个MIMD计算机上的算法一般会被拆分为一系列的子问题，每一个子问题都在MIMD机器的处理器上执行。通过给MIMD机器里的每一个处理元件同样的程序去执行，MIMD机器就可以被看作是一台SIMD计算机了。MIMD计算机的粒度较SIMD机器的粒度小得多。MIMD计算机倾向于采用数量少但处理能力很强大的处理器，而不是像SIMD计算机那样采用大量处理能力不很强大的处理器。

MIMD计算机可以是两种类型之一，这两种类型分别为共享存储器MIMD（Shared-Memory MIMD）和消息传递MIMD（Message-Passing MIMD）。共享内存MIMD系统有一个高速处理器阵列，每一个处理器都有局部存储器或缓存，并且每一个处理器都可以访问一个大容量的全局存储器（如图1-9所示）。这个全局存储器包含着将由该机器执行的程序和数据。也就是在这个存储器里，还有等待执行的处理数据操作（或子程序）表。每一个处理器将从该表中取一个数据处理操作和相关数据到其局部存储器或缓存中，并将半自治地运行系统中其他的处理器。数据处理操作间的通信也是通过这一全局存储器发生的。

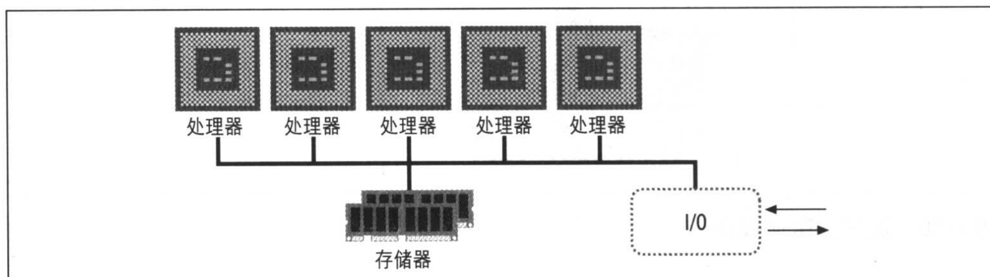


图1-9：共享存储器MIMD

在几个强大的处理器之间通过共享程序可以获得处理速度的优势。然而，系统的逻辑必须对访问共享存储器和相关共享总线的处理器作出仲裁。此外，处理器试图访问全局存储器里已废弃数据的这种情况，也必须得到允许。如果处理器A读取一个处理数据操作和数据结构到其局部存储器里，随之对该数据结构进行修改，那么试图访问主存储器中同样数据结构的处理器B就必须获知这一数据结构的新近版本的存在。这样的仲裁在像Motorola MC88110（现已淘汰了）这样的处理器里得以实现，Motorola的这一处理器就是用于共享存储器MIMD机器的。

另外一种可选的MIMD体系结构是消息传递MIMD（如图1-10所示）。在这种系统中，

每一个处理器都有自己的局部主存储器，而没有全局存储器。这种机器里的每一个处理元件（带有局部存储器的处理器）要么加载要么被加载将要执行的程序（及其相关数据）。每个数据处理操作自治地运行在局部处理器上，数据处理操作之间的通信经由公共媒介通过消息传递来实现。处理器之间的通信可以通过一个单一共享总线（如Ethernet、CAN或SCSI）来实现，或者通过采用一个更为精致的处理器间连接体系结构（如2-D阵列、N维超立方体、环型、星型、树型或完全互连系统）来实现。

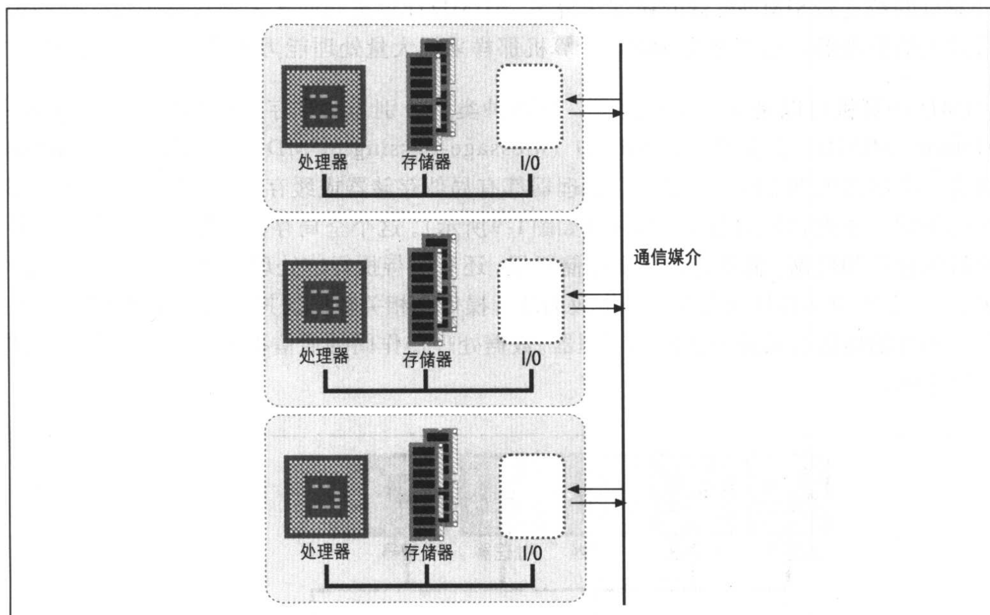


图 1-10：消息传递 MIMD

这种体系结构的机器不必受困于共享存储器机器的总线连接问题。然而，这种消息传递MIMD机器的处理节点的最有效和高效的互连方式仍然是一个主要的研究领域。每一种不同的体系结构都有其优点，究竟哪一种体系结构最适合于给定的应用，在一定程度上取决于应用本身的性质。只需有限数量的数据处理操作间通信的问题可以在一台不具备高效互连性的机器上高效运行；反之，其他需要进行消息传递的应用可能由于通信媒介的问题而无法发挥作用。如果一个处理节点要花一部分时间用于为相邻节点进行消息通路设定，那么具有高效处理数据操作之间通信但具有很低互连性的机器，就可能花费较多的时间用于消息传递而较少的时间用于实际计算。

理想的互连体系结构是，每一处理节点都与任何其他处理节点直接通信连接的完全互连系统。然而，由于价格和高度互连性维护的原因，这种理想体系结构就不一定现实了。这个问题的解决办法是，为这一机器内的每一处理元件规定有限数目的连接；这种解决

办法基于这样的假定：一个处理元件可能没有必要或者也不可能与机器内的其他处理元件同时进行通信。这些来自每一处理元件的有限连接就可以使用一个纵横交换机来进行互连，从而通过每个节点的有限数目的连接来为整台机器提供完全互联性。

分布式机器是由单个计算机组成的，通过网络互连松散耦合成MIMD并行机。诸如Beowulf甚至SETI@Home这样的项目就可以看作是MIMD机器。分布式机器在嵌入式领域比较普遍。一个个小型处理节点分布在整个工厂里，提供了局部监测和控制，它们集合在一起组成执行一个全局控制算法的并行机器。一些商业和军用航行器的航空电子设备也是分布式并行计算机。

现在让我们来看一下计算机应用以及这些应用是如何与机器的体系结构联系起来的。

嵌入式计算机体系结构

计算机能用来干什么？它必须完成什么任务？它是如何与人和其他系统交互的？这决定了这台机器的功能，进而决定了其体系结构、存储器和I/O。

如图1-11所示，一台专用的台式计算机（不一定就是PC）有很大的主存来支持操作系统、应用软件和数据，以及一个到大容量存储器设备（硬盘、DVD/CD-ROM等）的接口。这种台式计算机带有各种各样的I/O设备，以便于用户输入（键盘、鼠标和音频输入）、输出（显示器及音频输出）以及互连（网络和外设）。快速的处理器需要一个系统管理器来监控其核心温度、供给电压以及进行系统重启。

大规模嵌入式计算机也可能采用上述的形式。例如，它们可能作为一个网络路由器或网关，从而需要一个或多个网络接口、大容量内存以及快速操作。它们也可能需要某种形式的用户界面来作为嵌入式应用的一部分；在许多情况下，它们也可能只是专用于某一特定任务的常规计算机。因此，单就硬件而言，许多高性能嵌入式系统与常规台式机没有什么大的差异。

较小的嵌入式系统使用微控制器作为它们的处理器，这样做的优点在于处理器可以将很多计算机功能包含在一个芯片上。一个基于普通微控制器的专用嵌入式系统如图1-12所示。

微控制器至少有一个处理器、一个小容量的内部存储器（ROM和/或RAM），以及作为子系统模块在一个微控制器内部实现的某些形式的I/O。这些子系统为处理器提供了附加功能，许多处理器通常都有这种子系统。你经常可以在微控制器里发现这些子系统，这些子系统将在以后的章节里讨论。现在让我们快速浏览一下这些子系统，看看使用它们的目的何在。

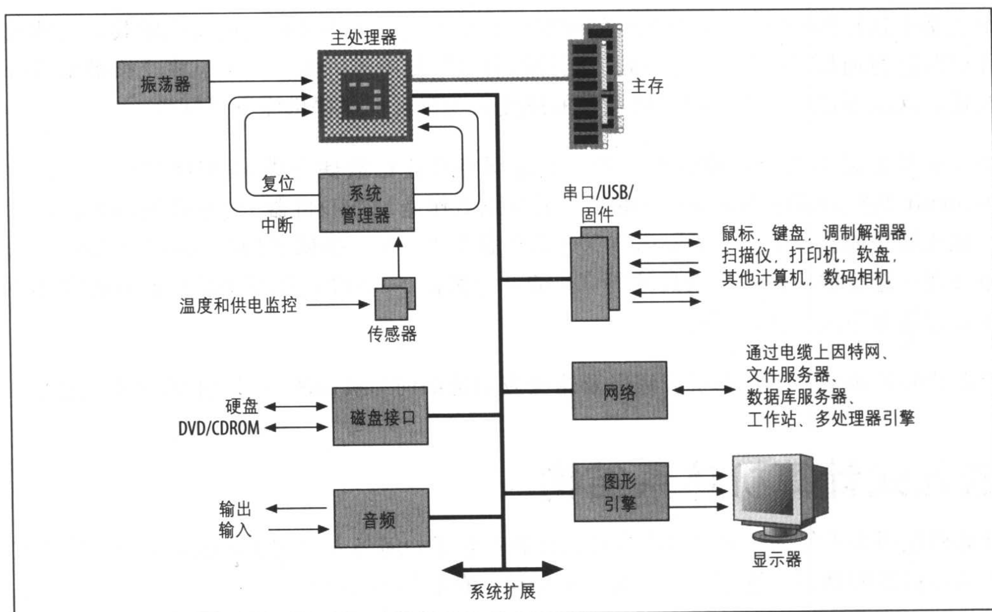


图 1-11：普通计算机框图

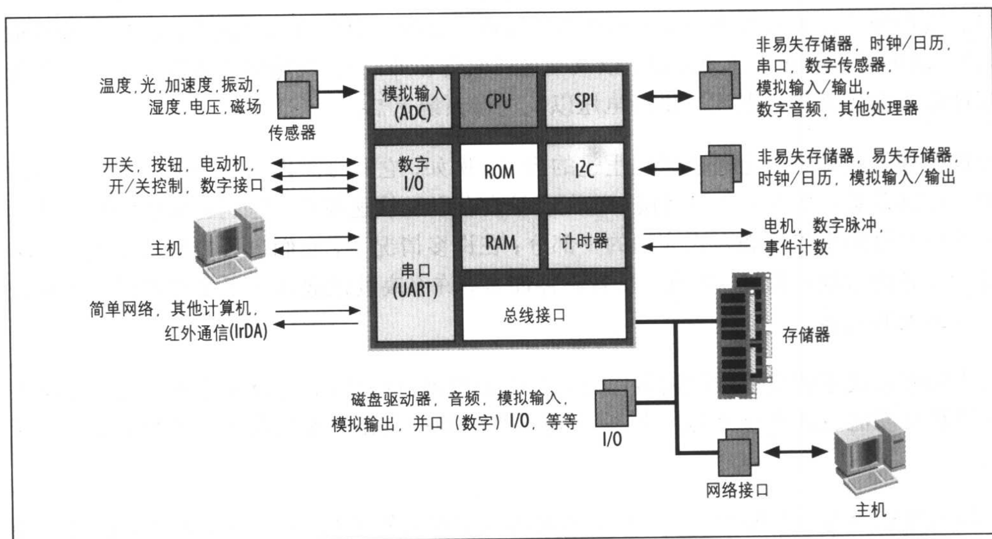


图 1-12：嵌入式计算机框图

最常见的I/O是数字I/O，它们都是端口，可以以引脚为基础，由软件配置成数字输入或者数字输出。作为数字输入，这些端口可以用来读取开关或按钮的状态以及读取另外一

个设备的数字状态；作为数字输出，它们可以用来开启或关闭外部设备，也可以向外部设备传达工作状态。例如，一个数字输出端口可以为一个电机激活控制电路，开/关灯，或者可能的话还可以触发诸如花园灌溉系统的水压阀这样的设备。将数字输入/输出端口结合在一起，可以用来合成对其他芯片的接口和协议。除了数字I/O之外，大多数微控制器还有其他子系统。而如果不需要其他子系统的功能，那么微控制器本身也能将其他子系统转换到通用数字I/O。作为一个系统设计者，这就为你在所设计的应用里如何使用微控制器提供了多种选择。

许多微控制器也有模拟输入，允许为监控或记录目的而对这些传感器进行采样。这样，一台嵌入式计算机就可以用来测量光强、温度、振动或加速度，大气或水的压力，湿度或者磁场之类的项目。或者，这些模拟输入可以用来监测简单的电压，或许用来确保较大规模系统的可靠操作。

有些微控制器有串口，通过串口，嵌入式计算机可以与一台主机、另外一台嵌入式计算机或者也许是一个简单的网络进行连接。诸如SPI和I^2C这样的专用形式的串口为扩展微控制器的功能性提供了一种简单的方法。这些串口允许外部设备与微控制器连接起来，为诸如片外存储器（用于数据或参数存储）、时钟/日历芯片、带数字接口的传感器、外部模拟输入或输出，甚至音频芯片及其他微处理器提供了访问途径。大多数微控制器带有计时器和计数器，用来在固定的时间间隔里为多任务处理产生内部中断，为片外系统提供外部触发，或者为电机提供控制脉冲，或者用来为来自其他设备的外部触发（脉冲）计数。有少数的微控制器还包含诸如USB、以太网或CAN这类的网络接口。本书中，我们将会详细介绍许多这类外部子系统，以及如何使用它们来增强嵌入式计算机的性能。

一些比较大的微控制器还提供一个总线接口，将内部地址、数据和控制总线展现给外部世界。这就使得微处理器能够以与常规处理器近乎同样的方式与大量可能的外部设备进行连接。以前所描述的所有可能的设备和接口都可以通过总线接口和适当选择的外部设备来实现。总线接口提供了无限的可能。

各个微控制器具有的I/O子系统的组合差别相当大。有些微控制器只针对简单的数控应用，可能只含有数字I/O；其他的一些微控制器可能面向工业应用，因此可能有数字I/O、模拟输入、电机控制以及网络连接等。微控制器（毫不夸张地说，具有来自众多厂商的几千个种类）的选择依赖于你的处理需要和接口需求，请选择最适合的一款。

第2章

汇编语言

对于学过才会做的事情，
我们是在做的过程中学会怎么去做的。
——亚里士多德
《人类伦理学》

本章是关于编写汇编语言（Assembly Language）软件的。这是一个很不容易介绍的主题，因为汇编语言本身变化多样。本书的内容涵盖了多种处理器。不同处理器的汇编语言是不相关的。每种处理器的汇编语言需要一本书（或数本书）的篇幅才能说得清楚。因此，我并不打算介绍每种处理器的指令集。相反，我将把重点放在68HC11和PIC这两种处理器上，以此介绍汇编语言的若干基本技巧。本章并不会详细剖析处理器的指令集，相关细节可以参考处理器的技术手册或用户手册。

汇编语言的编程工作于机器层上，以至于好像你真的知道处理器在做什么以及计算机实际上如何工作一样。尽管汇编语言编程可能会很有趣，但是若用它来编写主要的应用程序，那将是一项艰巨的工作。因此，汇编语言通常只应用于两种场合。一种是在早期的系统原型开发过程当中编写简单的测试软件。这类软件可能只需把机器初始化成一个已知状态，并进行一些简单的工作，比如让LED不断闪烁，或是将“Hello，World”字符串送往终端或主机。

使用汇编语言，你可以一次以一个字节或一个字来处理数据。处理器会把数据存储在内存中，这通常用计算机系统中的外部芯片来实现。在要处理数据时，处理器才会把数据从内存读到CPU内部的寄存器。

世界上没有哪台计算机可以直接解理解汇编语言。回想过去靠蒸气驱动、由专人照看计算机的时代，软件都是靠手工来编译的。每一个指令助记符都是由程序员查找并转换成

适当的操作码。虽然代码也是由字符构成的，但是把汇编代码转换成操作码确实很麻烦，尤其是对于大型程序更是如此。为了使编程变得容易，一种被称为汇编器的专用编译器便承担起从汇编助记符到操作码的转换功能。

汇编语言通常被描述成“低级语言”(Nuts-and-Bolts Language)，因为程序员是直接在为处理器编写代码。因为程序员可能要写大量的软件，所以会选择像C语言这样的高级语言。高级语言使得软件开发更容易，并且所编写的代码在不同的目标机器之间具有某种程度的可移植性。这些高级语言编译器将源代码直接转换成机器的操作码。因此，通过使用高级语言编译器，程序员便得以从不得不了解处理器的细节以及直接以机器码编写程序中解脱了出来。

因此，我们有足够理由来使用高级语言。但是，在许多时候程序员还是直接用汇编语言来写程序。为什么呢？由于汇编语言和机器码是“亲手写的”，因此能够被调节到发挥处理器和计算机硬件的最大性能。这对于I/O设备操作时间要求严格的处理而言，是极其重要的。此外，直接用汇编语言写的代码有时候（并非总是）所需代码空间较小。因此，如果你试图在有限的存储器里存放复杂的软件，并要求软件运行得迅速高效，那么，汇编语言可能是你最好（并且唯一）的选择。当然，其缺点就是你所编写的软件维护难度很大，并且几乎没有任何移植到其他处理器的可能性。一个好的软件工程师能够编写出较一般C语言编译器更为高效的代码，然而，一个好的编译器也可能产生出比一个普普通通的汇编语言编写得更为紧凑的代码。典型的做法是，可以在你的C语言代码中内嵌汇编语言，从而达到两全其美的效果。

许多习惯于其他编程语言的固执的程序员只要稍微提及汇编语言就会害怕，如入虎穴。但是实际上汇编语言编程并没有那么难，而且经常充满乐趣。把汇编语言和处理器视为一体吧。

前面已经说过，本书是主要讨论硬件而非软件的书，所以在本章中，“汇编语言编程”这个主题我们只是点到为止。在下一章里我们将会看到被广泛应用于嵌入式系统中的Forth编程语言。对于打算进一步了解嵌入式软件开发的读者，我极力推荐O'Reilly出版的两本书，分别是Michael Barr和Anthony Massa合著的《Programming Embedded Sysems》以及Mike Loukides和Andy Oram合著的《Programming with GNU Software》。如果你要构建基于Linux的嵌入式系统，我推荐O'Reilly出版的另外一本书：Karim Yaghmour所著的《Building Embedded Linux Systems》（该书中文版《构建嵌入式Linux系统》已由中国电力出版社出版）。

现在让我们来看一下处理器内部的存储器件。

寄存器

寄存器是处理器的内部（工作着的）存储器，不同的处理器体系结构之间寄存器的数量差距极大。典型的情况是，处理器含有一个或多个累加器。所谓的累加器就是可以在其上进行算术操作的寄存器。在一些处理器体系结构中，所有的寄存器都具有累加器功能，而在另外一些体系结构中，有些寄存器只用于数据存储而具有有限的功能。

一些处理器具有索引寄存器，能够通过它指向内存空间。在一些处理器体系结构中，所有的通用寄存器都可以作为索引寄存器；而在另外一些体系结构中，有专门的索引寄存器。

所有的处理器都有一个程序计数器（也叫作指令指针），它跟踪将要被读取并执行的下一条指令在内存中的单元地址。所有的处理器都有一个状态寄存器（也叫作条件代码寄存器或CCR），其中包含反映当前操作状态的各种状态位（也叫作标记）。这些标记可以表明最近一步操作的结果是否为零或为负，是否产生了进位，是否接受一个中断，等等。

有些处理器里有一个或多个控制寄存器，里面包含了影响处理器操作的配置位以及各种内部子系统的操作模式。许多外部设备也有一些寄存器，这些寄存器控制着外部设备的操作或者包含外部设备的操作结果。这些外部设备的寄存器通常被映射到处理器的地址空间中。

有一些处理器含有一些影子寄存器（shadow register），其中保存了当处理器进行一个中断服务时主寄存器的状态。通过使用影子寄存器，就不需要把处理器的状态保存在堆栈中。这样可以在进行中断服务时节省时间（稍后将详细讨论堆栈操作。）

处理器通常所说的8位、16位、32位或者64位，是指它们的寄存器的位宽。8位处理器价位较低，适合于相对简单的控制和监控应用。如果需要更强的数据处理，那么位宽较高的处理器会更好一些，虽然价格和系统复杂性也将相应升高。

机器码

处理器所处理的任何东西都会被表示成存放在存储器中的数字。这些数字可能是数据，也可能是软件。C程序经过编译之后会被转换成一些对处理器有意义的数字，处理器会将其视为一系列的指令。一个处理器（以68HC11为例）所能够执行的程序看起来如下所示：

```
86 41 B7 01 00 BD 02 00
```

这就是所谓的机器码（machine code），每个数值形式的指令又称为操作码（opcode）。对于人类而言，此类程序既难写又难以理解。为了化繁为简，我们使用称为汇编语言（assembly language）的标识法。汇编语言指令可以直接对应到相应的机器码。既然机器码对人来说难读又难写，我们就使用助记符（mnemonic）来表示操作码。例如，“86 41”这个68HC11指令转换成它的助记符“LDAA #$41”（$在某些汇编语言中代表十六进制的数值）会比较好理解。不过，这仍然会让人有些困惑，所以通常在程序的右边加上注释，避免我们以后忘了该指令的作用。前面的机器码序列若写成68HC11汇编语言，就会变成如下所示：

```
机器码          汇编语言          注释
86 41           LDAA  #$41        ; 分号之后的任何内容
B7 01 00        STAA  $0100       ; 都是注释
BD 02 00        JSR   $0200       ;
```

其中“LDAA #$41”表示“把立即数（#标识）0x41载入累加器A”。所谓立即数，意味着操作数可以被当作数值（而不是指向数值的地址）来处理。有些处理器将此称为常量值（literal）。LDAA对应机器操作码86。

“STAA $0100”用来要求处理器把累加器A中的内容存储到内存地址（这次没有#）0x0100。

“JSR $0200”表示“跳转到地址为0x0200的子例程处执行”。子例程本质上是一个程序或一个函数调用。在某些处理器中，则以调用（call，而不是jump）作为这条指令的助记符。

Freescale Semiconductor 68HC11系列处理器（见第16章），其指令长度为1~3个字节。其中第一个字节始终为操作码，其后是长度为一个或两个字节的操作数。例如，下面这段汇编语言程序：

```
机器码      汇编语言        注释
CE 10 02    LDX #$1002      ; 将0x1002载入X索引寄存器
5F          CLRB            ; 清除B累加器
86 41       LDAA #$41       ; 将0x41载入累加器A
```

注意每条指令的长度，第一条指令为3个字节，第二条指令为1个字节，而第三条指令是3个字节。一条指令所需的字节数取决于它所进行的运算。对于CISC处理器而言，这点没错，但对于RISC处理器来说则不然。为了简化指令解码单元，RISC处理器采用了固定长度的指令，将操作码包含在指令里。例如，PIC16处理器中每条指令的长度均为14位，而与所进行的运算无关。

不同类型的处理器将使用不同的汇编语言。前面的例子所使用的是68HC11的汇编语言。

该汇编语言只能应用在6800系列的处理器上，而68HC11便是此系列处理器的成员之一。其他汇编语言因为是基于不同的处理器硬件，所以具有不同的语法。表2-1中针对相同的运算（把一个值为0x41的字节载入一个寄存器）列出了各种汇编语言的语法（还要注意表达十六进制数的不同方式）。

表2-1：各种汇编语言的差异

处理器	指令	
Motorola 6800/68HC11	LDAA	#$41
Intel X86	MOV	al, 41H
Motorola 68X0	move.b	#$41,DO
PIC16xx	movlw	0x41
Motorola 56000	move	#$0041,A
Intel 80960	lda	0x41,r4

有符号数

在深入讨论汇编语言的相关细节之前，应该先了解一下机器内部如何表示有符号数。在二进制（或十六进制）数字系统中，一个数到底是负数还是正数，取决于最高有效位的状态。如果最高有效位是1，则表示为负，否则为正。

注意： 是否将一个寄存器或内存单元中的内容视为有符号数或无符号数，由程序员自己决定。在C语言中，我们习惯于显示地声明一个变量或指针是signed还是unsigned，而在汇编语言里则由你决定。

一个宽度为8位的累加器（以十进制来说），其数值范围可以为0~255（无符号数），也可以为-128~+127（有符号数）。对一个正数取二的补码就可以得到其负数。具体做法是：先将每个位的状态取反（取1的补码），然后在结果的最后一位加1（译注1）。因此，0xFF就是1111 1111（二进制），是一个负数。0xFF表示-1，0xFE表示-2，以此类推，0x80表示十进制的-128。

例如，把-3转换成一个8位的十六进制数，该这么做：

```
         0000 0011 (0x03)
按位取反  1111 1100 (0xFC)
结果加1   1111 1101 (0xFD)
```

译注1： 正数的补码与其原码相同；而负数的补码也可以通过将其原码中除符号位之外的其他各位取反加1得到。

所以，0xFD就是-3的十六进制表示。

请注意，我们并不一定要把一个数值当成有符号数（正数或负数）来看。我们可以根据自己的需要忽略正负号位，把该数当成无符号数来看，因此它的数值范围就是0x00~0xFF（即十进制的255）。

对于16位、32位以及64位的有符号数，其道理一样，最高有效位代表符号位，而其余位代表数值。

寻址模式

一条指令访问寄存器或内存单元的不同方式，叫作处理器的寻址方式。不同体系结构内的可用寻址方式的类型有所不同。让我们以68HC11体系结构的处理器为例来说明最基本的寻址方式。

寄存器寻址

指令只对寄存器内的操作数进行处理。例如用CLR B指令清空累加器B。

立即数/字面寻址

指令本身携带的一个常量数值作为操作数。

直接寻址

指令直接访问短地址指定的内存单元。换句话说，直接寻址方式能够访问整个地址空间中的一个子集。例如，在一个16位地址总线的处理器上，直接寻址方式能够确定第一个256字节内的一个地址单元；在一个32位地址总线的处理器上，直接寻址方式能够确定内存中第一个64K内的一个地址单元。如果可能，直接寻址就被用来减少访存指令的长度。这样就可以降低代码大小，进而在时间要求严格的应用中缩短取指时间。许多处理器，尤其是RISC，却并不支持直接寻址。

扩展寻址

在此模式中，指令通过指定完整的地址来访问内存单元。因此，指令“LDAA $B098”就是把内存单元0xB098的内容加载到累加器A中。

索引寻址

在此寻址模式中，指令把寄存器中的内容作为指向内存单元的指针。例如，假定X寄存器的内容为0x5034，那么指令“LDAA 0, X”表示“把X寄存器所指向的内存单元（此例中为0x5034）的内容载入到累加器A”。而“LDAA 2, X”则表示“先将X寄存器的内容加上偏移量2，那么其结果就是0x5036”。所以该指令就会从内存中地址为0x5036的内存单元载入数据。当你的程序需要引用表格或

列表数据时，索引寻址将会很有用。通过该寻址方式，程序员可以在数据结构中游刃有余地处理数据。

相对寻址

该寻址方式中有一个偏移量作为地址的一部分，例如分支指令就使用相对寻址来为程序计数器增加（或减少）一个值。例如，指令“`bra 02`”会把2添加到程序计数器而`bra $FF`则会把-1加入程序计数器。注意，程序计数器总是指向下一条将要执行的指令，而不是当前所执行的指令。

Big-Endian 和 Little-Endian

微处理器的体系结构要么是*Big-Endian*，要么是*Little-Endian*。这要看处理器存储数据（16位或更多）的方式。Big-Endian处理器将高位字节放在低地址单元中，如图2-1所示。每一种情况下，数据都存储在地址`0x0100`中。

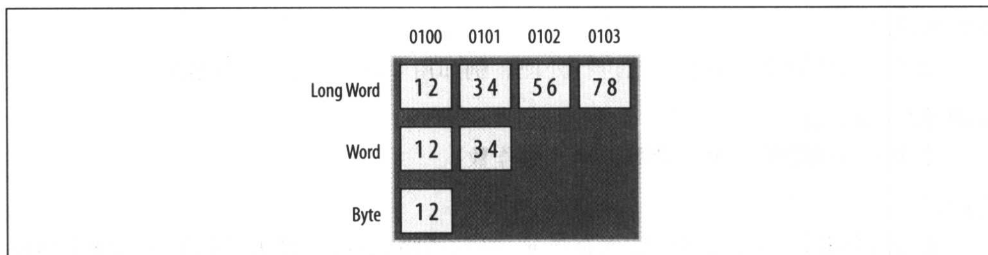


图2-1：Big-Endian

Little-Endian处理器将高位字节放在高地址单元中，如图2-2所示。

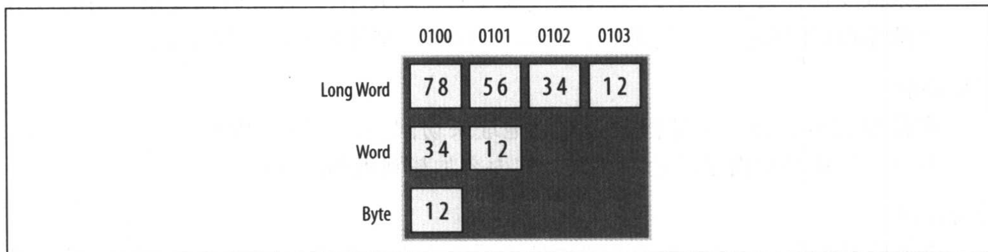


图2-2：Little-Endian

在Little-Endian图释中，低位字节沿着低位数据总线传输到低地址内存单元中。也就是说，对于程序员而言，从概念上很容易理解“数据通路”这一术语。Little-Endian的缺点在于，数据在计算机内存中以相反的顺序出现。例如，存储数值`0x12345678`到内

存中，会在内存空间中产生0x78563412这样的数值。值得注意的是，Little-Endian处理器能够正确地将这一数值读回，这只不过会令人们在直接查看内存单元中的数值时理解起来感到比较困难。相反，Big-Endian处理器把数值0x12345678以0x12345678的形式存储在内存单元中，这（对于人们来说）看起来更为合理。就操作而言，上述两种字节存储顺序中的任何一个都不比对方更具优势；它们只是做同样事情的两种不同的方式。当你在系统上进行高级编程时，"Endian意识"存在很少差异，毕竟你很少有机会暴露字节存储顺序。然而，当你开发和调试硬件以及底层固件时，将自始至终涉及到字节顺序，因此，理解Little-Endian和Big-Endian就比较重要了。

用汇编语言编程

汇编语言编程并不难，尤其是对于较小型的微控制器和微处理器体系结构。其主要的困难在于，许多人似乎已经知道如何解决所遇到的编程问题。就像任何其他编程语言一样，完成一个程序的方法不止一种，你很难找到一个绝对正确的方法（尽管它可能是最有效率的方法）。一个简单的解决方法就是把编程问题分解成一连串的小步骤，分而治之。

举例来说，假如我们要用68HC11处理器进行以下运算：把三个数0x10、0x1F和0x0C相加，把结果存到地址0x0027中。这个问题可以被分解成四个步骤。（第一步）取第一个数，（第二步）与第二个数相加，（第三步）与第三个数相加，（第四步）存储结果。当我们进行算术运算时，会选择一个累加器来保存这些数，因为累加器是专为此类运算而设计的寄存器。首先，我们会把第一个数载入到一个累加器：

```
LDAA   #$10;          将第一个数载入累加器A
```

累加器A现在包含数0x10（即十进制的16）。指令中的"#"符号表示该指令采用立即寻址方式。换言之，处理器会直接载入0x10这个数，而不是把它当作指针来处理。接着，我们打算对0x1F这个数执行加法运算，所以下一条指令为：

```
ADDA   #$1F;          将第二个数加到累加器A
```

这会使得0x1F被加到累加器A中。注意，这条指令还是用"#"符号。类似地，我们对第三个数做加法：

```
ADDA   #$0C;          将第三个数加到累加器A
```

最后，把结果存储到地址0x0027中：

```
STAA   $0027;         将累加器A的内容存入地址0x0027中
```

注意，这次并没有使用"#"符号。这是一个扩展指令。它表示：将累加器A中的内容

存到内存地址 0x0027 中。处理器不会把 0x0027 看作一个数，而会把它看作一个内存地址。

因此，整个程序如下所示：

```
LDAA  #$10          ; 将第一个数载入累加器 A
ADDA  #$1F          ; 将第二个数加到累加器 A
ADDA  #$0C          ; 将第三个数加到累加器 A
STAA  $0027         ; 将累加器 A 的内容存入地址 0x0027 中
```

要把这个程序作为子例程（可调用的函数或程序）来用，只需要在该程序的开头处附加一个标号，并在该程序结尾处附加一条 RTS（return-from-subroutine）指令（稍后我们深入讨论子例程）。编译器在编译期间会使用标号来确定所给定的地址单元。它在机器码中并不具备任何直接的意义。

```
add_numbers
  LDAA  #$10        ; 将第一个数载入累加器 A
  ADDA  #$1F        ; 将第二个数加到累加器 A
  ADDA  #$0C        ; 将第三个数加到累加器 A
  STAA  $0027       ; 将累加器 A 的内容存入地址 0x0027 中
  RTS               ; 返回主程序
```

要从主程序中调用子例程，只需要把标号作为操作码调用指令 JSR（Jump-To-Subroutine）即可：

```
JSR  add_numbers
```

相同的子例程用 PIC16 汇编语言来写，看起来会如下所示：

```
add_numbers
  movlw 0x10        ; 把第一个数载入累加器 w
  addlw 0x1F        ; 把第二个数加到累加器 w
  addlw 0x0C        ; 把第三个数加到累加器 w
  movwf 0x27        ; 把累加器 w 的内容存到地址 0x27 中
  return            ; 返回程序
```

movlw的作用是“把字面值移到累加器 w”，addlw的作用是“把字面值加到累加器 w”，movwf 的作用是“把累加器 w 的内容移到寄存器文件（f）”（Microchip 公司把处理器芯片之内的 RAM 称作寄存器）。PIC 处理器的 return 指令实际上与 68HC11 的 RTS 指令没有什么区别，两者的功能完全一样，只不过是助记符不同罢了。

要调用 PIC 子例程，只需调用 PIC 的 call 指令，并以标号作为操作码：

```
call add_numbers
```

要将此函数转换成机器码，可以通过指令集的数据手册来查找每条指令，或者在主机上

执行编译操作。让我们以手工的方式来编译68HC11版的子例程，以便让你看到汇编语言编译的整个过程（操作码用十六进制来表示，其前端部分代表内存地址）。

LDAA（立即寻址）指令的长度为两个字节，操作码是86，另外一个字节是操作码。而ADDA（立即寻址）指令的长度也是两个字节，操作码为8B。在技术手册或程序员参考手册中查找到其余的操作码，最后就会得到如下的结果：

```
0100   86    10     LDAA #$10          ; 把第一个数载入到累加器A
0102   8B    1F     ADDA #$1F          ; 把第二个数加到累加器A
0104   8B    0C     ADDA #$0C          ; 把第三个数加到累加器A
0106   B7    00 27  STAA $0027         ; 把累加器A的内容存到地址0x0027中
0109   39    RTS                       ; 返回主程序
```

此处，我（随意地）作了如下假定：程序的起始地址是0x0100，每条指令的地址都附在最左边。注意地址的变化与指令长度的关系。

如果我们要明确地告诉编译器，程序的起始地址是0x0100，则可以使用org这条伪指令语句：

```
org 0x0100
```

org伪指令必须始终放在第一条指令的前面，用于把程序的代码定位在特定的位置上，例如处理器芯片的ROM上。

现在让我们以手工的方式单步执行程序代码，看看处理器是如何执行该程序的。假定程序计数器指向地址0x0100，所以处理器第一步先载入地址0x0100中的内容，并把其当作指令来解码。处理器通过把0x0100放到地址总线上开启一个读周期来实现这第一步。存储设备的响应是把地址0x0100的内容（86）送到数据总线上。然后由处理器把该字节读到指令解码单元。86就是LDAA（立即寻址）指令，这会使处理器载入该指令的下一个字节（在0x0101中），并把它读到累加器A。在处理器里，指令解码的另外一种操作是，把程序计数器（加上当前指令的长度）指向下一条指令。程序计数器的值变成0x0102，因为第一条指令的长度有两个字节。

处理器为了载入位于0x0102的指令，会把该地址放到地址总线上，并执行一个读周期。处理器会把它解码成ADDA（立即数寻址）指令。然后处理器读取下一个字节，并把它加载到累加器A。同样地，处理器会执行下一条ADDA指令，并把位于0x0105的字节加载到累加器A。

程序计数器现在指向地址0x0106。处理器为了读取这条指令，会把0x0106放到地址总线上，并执行一个读周期。存储设备的响应是，把B7放到数据总线上。处理器把B7解码成延伸存储操作。为了载入后面两个字节，处理器会先把0x0107放到地址总线上，

执行一个读周期，然后把0x0108也放到地址总线上，执行另外一个读周期。处理器会把读取的这两个字节组合成0x0027。然后，处理器把0x0027送往地址总线，把累加器A的内容放到数据总线上，继而执行一个写周期。这样，地址译码器所对应的存储器芯片或外设就会被锁存并存入所写的数据。

正常情况下，每执行一条指令，程序计数器就会被加1，所以它现在会指向下一条指令。有些指令，如跳转（JMP）或分支（BRA）会直接修改程序计数器的值，让我们控制程序的执行流程。这些指令会把一个新的值载入到程序计数器（如跳转指令）或是把一个值加到程序计数器（如分支指令），所以程序计数器会指向不同的内存单元。这意味着将要执行的下一条指令来自新的地址，而不是当前执行的指令的下一个地址。

反汇编

反汇编就是把一系列的机器码转换成代表该程序的助记符。如果我们拿到一份（可能由他人编写）机器码程序，而我们想要知道它的功能以及它的工作方式，可以对它进行反汇编。

例如，假设我们拿到一份68HC11机器码程序的字节序列如下：

```
8E 56 78 86 56 84 06 36 4C 36
```

首先，我们假定第一个字节为操作码。在Motorola 6800（或68HC11）指令集的数据手册里，查找0x8E这个操作码，你会发现它是LDS（load stack pointer，加载堆栈指针）指令，而该指令的长度为3个字节（一个字节为操作码，另外两个为操作数）。因此，如果第一个字节为操作码，则其后的两个字节为相关的数据。所以，第一条指令就是：

```
8E 56   LDS #$5678;          把0x5678载入到堆栈指针
```

既然第一条指令为3个字节，那么从第4个字节开始应该是下一条指令。因此，下一个操作码是0x86（也是根据指令集的技术手册查到的），它就是LDAA #（把一个立即数载入到累加器A）指令，该指令的长度为2个字节。

所以现在反汇编的结果是：

```
8E 56 78     LDS #$5678          ; 把0x5678载入到堆栈指针
86 56        LDAA #56            ; 将0x56载入到累加器A
```

下一个操作码是0x84，它就是ANDA指令，而且该指令的长度为2个字节。接着，我们就会查到0x36（PSHA）、0x4C（INCA）以及另一个0x36（PSHA）。

所以，我们可以把机器码：

```
8E 56 78 86 56 84 06 36 4C 36
```

反汇编成：

```
8E 56 78     LDS #$5678      ; 把0x5678载入到堆栈指针
8E 56        LDS #$56        ; 把0x56载入到累加器A
84 06        ANDA #$06       ; 对累加器A与0x06做AND运算
36           PSHA            ; 把累加器A压入栈
4C           INCA            ; 递增累加器A
36           PSHA            ; 把累加器A压入栈
```

位置无关代码

当我们的程序要跳转到某个地址或子例程时，可能会使用一个绝对地址。例如，JSR $0200会使程序跳转到绝对地址为0x0200的子例程。这意味着，该子例程必须始终处在地址0x0200，否则我们的程序将无法正常运行。一个程序（以及它的子例程）有可能被计算机的操作系统定位（或重新定位）在内存的任何位置。如果真的是这样，JSR $0200将再也无法跳转到子例程所在的位置，因为该子例程不再位于0x0200这个地址上。因此，程序就会崩溃。基于这个原因，最好是使用与位置无关的代码。方法是，在非必要情况下，避免使用绝对地址。这意味着，不应该跳转到子例程的绝对地址，而应分支到子例程在程序计数器中目前内容的相对位置。这听起来好像很复杂，不过它的实际意思是，我们应该使用BRA（或BSR）指令，而不是JMP（或JSR）指令。使用分支指令时，我们将一个数值加到程序计数器，而不是取代其中的内容。这也意味着，我们不会跳转到内存的某个绝对位置，而是分支到相对于当前位置的地方。

当我们所要分支或跳转的程序总是转到某个特定位置时，才使用绝对寻址方式。例如，一个（永久存储器）放在ROM里的子例程是不会移动到其他位置的。

循环

我们经常需要重复执行一条指令或一串指令。使用分支指令，我们可以重新开启程序计数器，从而达到控制程序执行流程的目的。下面的简单程序示例重复执行了15次：

```
        LDAA #$0F   ; 将计数值载入到累加器A
again               ;
        DECA        ; 把累加器A减1
        BNE again   ; 重复执行，直至倒数为0
```

DECA会把累加器A减1，并且适当地设置状态寄存器的标志位。如果累加器减1的结果为零，则CCR（Condition Code Register，条件代码寄存器）中的Z（Zero）标志位就会被设定；否则，该标志位就会被清零。BNE指令的功能是“不等于零就进行分支操作

(Branch not equal to zero)”。可以用来检查Z标志位的状态。如果该位被清除（不等于零），就会进行分支操作；否则就会继续执行BNE的下一条指令。因此，上面的程序段将会从累加器A等于15开始，重复执行减1操作，直到其值为零。在此状态下，循环就会终止。为了使循环得以运行，我们需要将重复执行的程序代码放在again标签之前DECA之后。

以机器码来看，上述代码就会像下面这样：

```
地址        操作码              汇编语言
0100        86 0F               LDAA #$0F
0102        4A           again  DECA
0103        26 FD               BNE again
0105        ...                 ;BNE的下一条指令的地址
```

注意BNE指令。0xFD是怎么来的呢？我们想让处理器分支到地址0x0102。对于所有其他指令，当BNE执行时，程序计数器会指向下一条指令，所以就此例而言，当BNE执行时，程序计数器就会变成0x0105。我们想让程序计数器变成0x0102，所以需要让程序计数器减去3。其方法是将程序计数器加上−3，也就是长度为8位的十六进制数0xFD（0xFF是-1，0xFE是-2，0xFD是-3，依次类推）。向下分支（branch forward）时，我们会加上一个正数；向上分支（branch backforward）时，我们会加上一个负数。不论上述哪种情况，在确定增加的数值之后，程序计数器都会指向要分支到的地址单元。

屏蔽

我们经常需要检查某些位（像外部设备的状态或标志）的状态。通常，我们只关心特定位的状态，而忽略我们所不知道（或是对我们不重要）的其他位。

举例来说，假如我们要检查一个字节（来自状态寄存器）中第0位的状态，此时，我们需要拿该字节与一个常量作比较。但是，我们应该选择哪一个常量呢？接下来的两个字节（来自状态寄存器）范例中，都设定了第0位。如果将两个字节与常量0x65作比较，则第一个字节可以成功对比，而第二个却不然：

```
01100101 (0x65)
11001101 (0xCD)
```

事实上，你无法找到一个可以对比任何字节的常数。所以在对比前，我们必须将其他位设定在某个已知状态。为达到此目的，需要屏蔽掉我们不感兴趣的其他位。我们可以用AND指令来达到此目的。

在前面的例子中，我们需要用0x01与读自状态寄存器的字节做AND运算，以便清除其

他位。这会保留第0位（不管其处于何种状态），而清除其他各位。如果第0位被设定，则AND运算的结果将会是0x01；如果第0位未被设定，则结果将会是0x00：

```
01100101 (0x65)
00000001 (0x01) AND
```

其运算的结果是：

```
00000001 (0x01)
```

接下来运算：

```
11001101 (0xCD)
00000001 (0x01) AND
```

结果如下：

```
00000001 (0x01)
```

而下面的运算：

```
10001100 (0x8C)
00000001 (0x01) AND
```

其结果为：

```
00000000 (0x00)
```

以上的每个范例中，原始字节中的第0位的状态都会被保留下来，而其他各位则会被清零。因此，为了判断第0位的状态，现在只需要将0x01与AND运算的结果作比较。如果相等，则代表该位被设定过；否则，代表该位未被设定。

我们把以上的说明转换成汇编语言。接下来的示例中我们会检查地址为0x0700的I/O设备的状态：

```
check LDAB  $0700          ; 将地址为0x0700的设备的状态寄存器读到累加器B
      ANDB  #$01           ; 屏蔽掉不需要的位
      CMPB  #$01           ; 检查第0位是否被设定过
      BNE   check          ; 如果未被设定，则返回继续检查
```

这个代码段循环会重复执行，直到第0位被设定。

索引寻址

经常需要用到指向一段内存的参考指针。68HC11具有两个索引寄存器：X和Y。它们两者等效于C语言的指针。

例如，我们想要把 0x60 填充到地址范围为 0x0200~0x2FF 的内存中。完成此操作的一个方法是：先将 0x60 放到一个累加器中，再把它存入地址 0x0200，接着存入地址 0x0201，等等：

```
LDAA #$60   ;
STAA $0200  ;
STAA $0201  ;
STAA $0202  ;
STAA $0203  ;
STAA $0204  ;
STAA $0205  ;
STAA $0206  ;      …就这样继续下去
```

尽管这样可以实现我们的目的，不过程序将变得非常冗长。一个较为简单的做法是使用索引寻址，也就是以此索引寄存器指向每一个内存单元。接下来的程序将会使用索引寻址来达到我们的目的：

```
     LDX #$0200        ; 把数值 0x0200 载入到寄存器 X
     LDAA #$60         ; 把所要存储的数值放到累加器 A
     LDAB #$FF         ; 把计数值存到累加器 B
loop STAA 0, X         ; 把累加器 A 存到寄存器 X（没有偏移量）所指向的地址
     INX               ; 修改寄存器 X 以指向下一个地址
     DECB              ; 倒数计数
     BNE loop          ; 重复执行，直至结束
```

LDX指令用来把一个立即数载入宽度为16位的寄存器X。它现在指向我们用来指向内存的指针。接着，我们把即将存到内存的数值保存到累加器 A。然后，把我们所要存取的内存位置载入累加器 B，这便是循环的计数器。

循环一开始会把累加器 A 的内容存到寄存器 X 所指向的内存单元。“0,X”代表没有偏移量（如果存放的位置并不是寄存器X所指向的内存单元，而是其他第三个位置，则我们可以使用“3,X”）。接着递增寄存器X以便指向下一个内存单元，然后递减累加器B，因为循环已进行一次。递减累加器的过程中，Z标志位可能会被设定或被清除，随后的分支指令将检查该状态。如果递减的结果不是零，则循环会继续下去。执行分支指令回到“STAA 0, X”指令，并把累加器 A 的内容存到下一个内存单元。上述过程会重复执行下去。

堆栈

许多处理器实现了一个或多个堆栈，将其作为外部存储器的临时存放地。处理器能够把一个来自寄存器的数据压入（PUSH）堆栈保存起来以备后用。通过出栈（POP）、处理器可以把该值从堆栈中取出返回给寄存器。在有些处理器体系结构中，出栈操作也叫拔栈。

大多数处理器都有一个称作堆栈指针的特殊寄存器，用于指向堆栈中的下一个空闲单元。有些处理器实现了不止一个堆栈，因此也就有多个堆栈指针。大多数堆栈沿着内存由高地址区向低地址区方向增长（也有一些堆栈沿着内存由低地址区向高地址区方向增长）。当处理器在堆栈中对一个值进行压入/弹出操作时，堆栈指针就自动地递减（或递增）以指向下一个空闲单元。

图 2-3 所示为当寄存器的内容（此例为 72）压入堆栈时所经历的步骤。这些步骤一般是由一条指令（例如 PSHA）来完成的。首先，从寄存器拷贝出该值，并把它存到堆栈指针所指向的位置。接着递减堆栈指针。再次提醒，这些步骤是由一条 push 指令自动完成的。

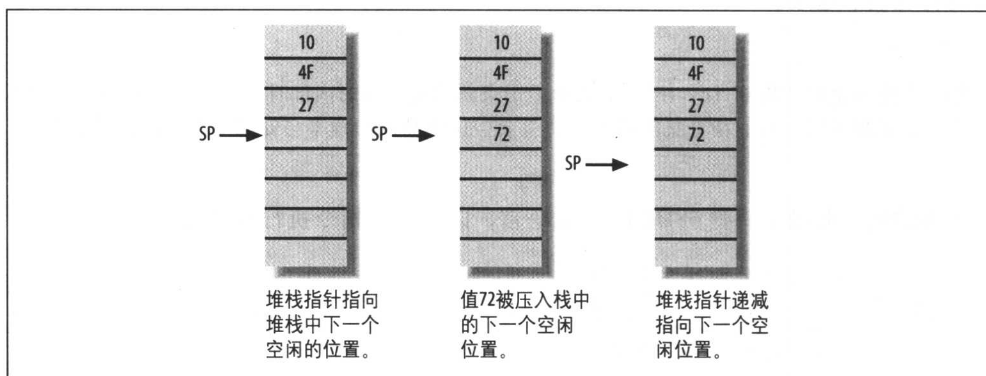


图 2-3：把一个寄存器内容压入堆栈

当诸如中断这样的异常发生时，大部分的处理器会自动把寄存器的内容压入堆栈。然而，程序员依然可以把堆栈用作临时存储器。有些处理器有两个堆栈，一个留给系统使用，另外一个给应用的程序员使用。

堆栈特别适合用来临时存储数据。例如，假设我们要调用一个子例程，但我们知道此子例程会改变累加器 B 的内容。如果在调用该子例程之前，累加器 B 并没有保存任何重要的数据，这将不会造成什么影响。但是，如果希望保留累加器 B 的内容，就需要在子例程被调用之前将累加器 B 的内容压入堆栈，然后在调用结束的时候取回该内容：

```
PSHB;        把累加器 B 压入堆栈
BSR fred;    分支到 fred 子例程
PULB;        从堆栈取回累加器 B 的内容
```

这样，我们就可以把堆栈当成数据（程序稍后将会用到的）的临时存放地。此外，一个好的程序员在编写代码时，应该自行保留任何可能受到影响的寄存器：

```
fred
```

```
PSHB   ;把累加器 B 的内容压入堆栈
       ;
       ;(此处为子例程的代码)
PULB   ;从堆栈恢复累加器 B
RTS    ;
```

指令的时序

在某些处理器上，尤其是 CISC 芯片，不同的指令完成工作所需要的时间也不一样。有时候，知道一段给定程序执行需要的时间是很重要的，尤其是如果该程序与对时间要求严格的外部设备交互时。程序执行时间的计算方法是，先求出每条指令所需要的周期数，然后将它们加起来。

注意：此处所说的“周期（cycle）”是相对于用来驱动处理器的内部时钟而言的。它的速度并不需要跟晶振一样。指令集的技术手册里会告诉你晶振的频率与处理器内部时钟的关系。

接下来的例子展示了一些 68HC11 汇编语言，以及每条指令执行时所需的周期数：

```
        LDAA #$0F    ; 2 个周期
again   DECA         ; 2 个周期
        BNE again    ; 3 个周期
exit    JMP $1000    ; 3 个周期
```

检查上述代码，我们会发现循环被执行了 15 次（0x0F）。循环执行一次总共要 5 个周期。因此，这段程序执行的时间为：

2+（15 × 5）+3=80 个周期

以 2MHz 的 68HC11 为例，一个周期要 500ns。因此，这段程序执行的时间为 40000ns（或 40μs）。

知道指令的时序是非常有用的。例如，假设我们的程序要提供大约 9ms 的延迟。为了达到此目的，我们需要将一个数载入到寄存器，并把它倒数至零。这将会为我们提供延迟时间。

因此只需要使用下面这段代码就可以完成：

```
        LDX #some_number   ; 把一个数载入到 X
repeat  DECX               ; 倒数 3 个周期
        BNE repeat         ; 倒数至零，则重复 3 个周期
```

对于 2MHz 的 68HC11，其周期为 500ns，所以 9ms 的延迟需要 18000 个周期。这个循

环每执行一次需要6个周期。因此，我们要执行3000次循环，其十六进制为0x0BB8。所以，上面的程序就会变成：

```
        LDX #0x0BB8  ; 把数值3000载入到X
repeat  DECX         ; 倒数3个周期
        BNE repeat   ; 倒数至零，则重复3个周期
```

现在，让我们来看几个重点。如果循环执行时发生中断，则处理器就会转而去执行中断服务请求，这就会为所设定的延迟增添不可知的变数。如果此程序正在处理某个紧急的事件，那么此类事件则会导致严重的后果。大部分处理器都会提供禁止或重新开启中断的指令，而你应该慎重考虑此程序执行之前停用所有的中断。当然，完成之后你还要记得重新开启中断。另外一个重点是，此程序必须在2MHz的68HC11处理器上执行，这样才会提供9ms的延迟时间。如果执行该程序的处理器内部时钟提升到2.1MHz，那么延迟时间就不再是9ms了。换言之，此类技术非常不具备可移植性，而且普遍被看作是个坏的编程习惯。所以，除非不得不使用，否则请不要这样做。使用此类程序的程序员，通常会在他们把代码移植到较新版本的硬件上时遭到惩罚，原本运行很好的程序会突然出问题。

一个让延迟时间固定的较好的办法是使用硬件定时器，这或者是微控制器内部的一个部件，或者是一个外部装置。此定时器会对处理器产生一个中断请求，让我们用作延迟时间的方法得以更精确、可靠并具有可移植性。

第3章

Forth与公开固件标准

对于聪明人，一句话就足够了。

本书开始便讲过，我想探讨的是硬件而非软件。市面上已经有很多不错的书籍讨论了C语言程序设计、如何以C语言来开发嵌入式软件、移植Linux、Python程序设计以及Java程序设计等等。然而，有一个议题读者却很难看到有书在讨论它，所以你也没有机会去了解它。这的确让人感到可惜，因为对于硬件开发人员而言，这是一个非常重要的议题。这个议题就是程序设计语言Forth。所以，我将会违背自己所说过的话，为大家介绍这种重要的程序设计语言。

Forth简介

Forth最初由Charles（Chuck） Moore（*http://www.colorforth.com/bio.html*）在1970年编写，主要用来控制位于美国基特峰国家天文台（Kitt Peak National Observatory）的30英尺的望远镜。在创建Forth的时候，Charles Moore预测“第四代（Fourth）”计算机将会是以嵌入式系统为基础的分布式控制器，尽管当时尚未发明该术语。因此，他称这种程序设计语言为Fourth。然而，因为他用来开发的机器只允许使用长度为5个字符的文件名，所以这个程序设计语言便被简称为Forth，这个名字一直沿用至今。

Forth不同于任何其他传统的编程语言（尽管它有点类似于Adobe的PostScript语言）。Forth是一个可扩展的、具有高度交互性的、以堆栈为基础的编程语言。这是一个非常有效率并且应用范围非常广泛的编程语言。此外，Forth可以非常容易地移植到另外一个全新的机器上，这个语言的结构和性能使得它非常适合用来调试系统软件和硬件。一个软件能用来调试新的硬件，这确实让人感到惊奇。Forth常用于尚在开发的系统上，而且厂家经常将其放到它们的计算机中。你经常会发现，Forth就隐藏在机器的ROM里。Forth

被 NASA 使用在航海家太空船（Voyager Spacecraft）上，以便提供有效率的固件，以及跨越广袤无垠的太空以交互的方式进行调试的能力。Forth 也被 NASA 使用在航天飞机（Space Shuttle）的恒星导航（Star Tracker）子系统上（更多信息可以在 *http://forth.gsfc.nasa.gov* 上找到）。此外，Apple Computer 和 Sun 公司以及其他的计算机厂商所使用的 Open Firmware 标准（译注 1）（IEEE 1275）也是以 Forth 为基础。Forth 备受硬件工程师的青睐，但传统的计算机科学家却对其嗤之以鼻。

注意： 如果你手头有一台苹果 Macintosh 机，而且喜欢冒险的生活，请在按下电源按钮后立即按住 command-option-O-F 键。这样，你就会进入 Open Firmware Forth 而不是 Mac OS X。

Forth 本身主要由 Forth 编写而成，而且在执行期间也是由 Forth 自己进行解释。为了提升执行的速度，也为了直接利用目标计算机的特性，用来执行 Forth 的虚拟机通常直接用汇编语言来编写而成。Moore 解释，他的 Forth 概念是每个软件都应该经过优化，以便充分利用它所运行的机器，即使这种优化的行为会对程序语言的可移植性大打折扣。这其中的哲理严重地违背了传统程序设计语言的理念。使用 Forth 时，讲究的是速度和稳定性，而不是平台之间移植应用程序的能力。

Forth 是编译器，也是解释器，也可作为操作系统。大多数计算机语言都具有共同特性，因为它们之中有许多来自共同的源头。此类语言包括C和C++、Pascal、Modula-2、ADA、Fortran 以及 BASIC 等等。除了 BASIC（典型的）是解释性语言外，其余均为编译式语言。Forth 也是编译式语言，不过它有一点不同，该语言具有交互性。命令（Command，在 Forth 中作为词，译注 2）在 Forth 中会立即获得回应。这个特点再加上 Forth 直接与硬件沟通的能力，使得它成为硬件开发的绝佳环境。

Forth 允许你单独执行程序中的任何子例程（词）。程序员可以根据先前所定的词来建立新的程序（词）。所以，你不必编写一个大型的程序，而可以分开编写并测试每段代码。由这些词组合成新的词，最后组合成一个词，也即整个程序。程序员也可以用这个新词来进行编程工作。就这样，程序员用来工作的语言不仅会增长，而且还变得丰富了。此外，这些用来建立新词的词仍可供程序员进行编程，而且是按照它们自己的方式来执行，或是用来建立新的词。这种形式的编程结构使得Forth成为一种“由下而上（bottom-up）”的编程语言，而非传统的“由上而下（top-down）”的编程语言。具有互动性以及底层的特性，使得 Forth 成为一种非常适合用来调试微处理器硬件的工具。（初期调试阶段）

译注 1：Firmware，即固件，存在于只读存储器中而不通过软件来执行的编程命令，例如存储为计算机指令编制的微程序的部件。Open Firmware 为公开固件标准。

译注 2：在 Forth 中，命令、子例程和程序都称作词（word）。

设计来与特定硬件交互的“词”，日后可以被并入较高层次的调试程序或最终的应用程序。

Forth最值得注意的地方，就是数据（参数）必须通过堆栈传递给词（如果你喜欢，也可以称之为子例程）。（请注意，此堆栈未必就是处理器自身的堆栈。通常它是Forth核心所建立的虚拟堆栈。）尽管许多语言也用堆栈来传递参数，不过程序员通常不知道此事。然而，Forth却完全不同。在Forth中，一个词所需要的参数必须先压入堆栈，然后才能调用。这些值也可以由其他词或者调用者压入堆栈，或从堆栈取出。

在Forth中，程序员所写的词可以直接操作堆栈，从堆栈中取出参数以及把新的值（返回参数）放回堆栈。此外，程序员也可以在提示符下直接操作堆栈，在所指定的词执行之前，手动压入或弹出堆栈。此特性包含了Forth的力量和美妙。由于使用堆栈，Forth的语法又被称作Reserve Polish Notation（保留波兰表示法，RPN）或者Postfix Notation（后标识法）。使用老式HP计算器的人都知道RPN。使用此标识法时，操作符被放在参数之后。所以，“*a*+*b*”这个传统算式在RPN中就变成了“*a b* +”。此例中，会先将a所代表的数压入堆栈，然后是*b*。在Forth中，“+”这个词会从堆栈中取出这两个数，执行加法运算，并把结果放回堆栈。

所以：

```
4 5 +
```

就会把9这个值放在堆栈的栈顶。要显示此结果，可以使用点号“.”。这个符号可以把栈顶的数取出来，并以有符号数的形式打印到控制台上。下一个例子是将三个数相加，并把结果打印到屏幕上去：

```
5 25 98 + + .
```

第一个“+”会把98与25相加，并把结果放到堆栈上。第二个“+”会把此结果与5加在一起。下面的表达式也可以得到相同的结果：

```
5 25 + 98 + .
```

Forth的大部分版本都是使用16位的有符号数，不过也有些版本明确指出所用的是32位。16位的Forth在使用32位的结果时会使用双精度。

在默认情况下Forth使用十进制来进行计算。`hex`可以把基数变为十六进制，而`decimal`可以把基数变回为十进制。例如，让我们把`0x4F`（hex）与`255`（decimal）相加，并把结果以十进制的形式打印出来：

```
hex 4f decimal 255 +
```

则在控制台上显示结果为334。

要将栈顶的数输出为无符号数，请用u，而不要使用“.”符号。要输出一个右对齐的数，可以用词“.r”，以栈顶的参数作为第二个参数指定所占的位数。例如：

```
123456  6 .r
1234    6 .r
12      6 .r
```

会输出下列结果：

```
123456
  1234
    12
```

标准的算术运算可以使用“+、-、*、/”。这些算术运算符其实是Forth所定义的词，所以它们在使用前必须加上空格。例如，2 乘以 3：

```
2 3 *
```

“/”运算符用来进行整除。不同的实现方法之间的主要差别在于，负数的时候，有些版本的Forth向零的方向取整数，有些版本的Forth向负无限大的方向取整数。例如，9 除以 3：

```
9 3 /
```

我们可以使用词1+（或1-）来递增（或递减）栈顶的数值。下面这个无意义的例子可以将 2 转换成 3：

```
2 1+
```

接下来要介绍的是（几乎）所有Forth实现中都会看到的一些“词”。我并不打算对这种语言作深入的探讨，关于该语言的完整讨论可以参考相关书籍。别忘了，在Forth中所有的词都必须以一个或多个空格隔开。

字符串

我们将以程序员所熟悉的“hello world”来说明在Forth中打印字符串的方式。用来输出字符串的词是“."”（念为 *dot quote*），因此应该这样：

```
." hello world"
```

注意“."”与字符串之间有空格，但是字符串与最后的引号之间却没有空格。这是因为“."”是一个词，而所有的词都必须以空格隔开。最后一个引号并不是一个Forth词，它只是一个字符串结束的标志。

Forth 词 cr 用来输出换行 / 换列符号，space 用来输出一个空格。如果要输出多个空格，可以使用 spaces，并在堆栈指定打算输出的空格数。例如，下面的程序会在控制台上输出 20 个空格：

```
20 spaces
```

要将一个数作为 ASCII 字符输出，可以使用词 emit。emit 等效于 C 函数 putchar。例如，如果要输出一个星号，可以这么做：

```
42 emit
```

Forth 词 key 会等待一个从控制台所输入的字符，key?(key query)会检查是否有按键被按下。注意，我曾看到过有些 Forth 的实现使用不同的词来等待从控制台所输入的字符。所以，你的情形可能会不一样。

词 expect 用于从控制台输入一个字符串。

警告：有些 Forth 的实现并不使用词 expect，而是使用词 accept。它们的工作原理是一样的。

---

expect 会用到来自堆栈的两个参数。最顶端的参数（最后被压入堆栈的参数）用来指定要从控制台接收多少个字符。expect 会一直等到所键入的字符达到这个数目，或是遇到回车符为止。expect 用到的另外一个参数是一个指向 scratchpad（暂存器）的指针。scratchpad 位于内存中的某个位置，用来暂存从控制台接收的字符串。但是，要将字符串放到什么地方呢？很简单，Forth 已经为你准备了一个 scratchpad。使用 pad 这个词就可以将 scratchpad 的地址放到堆栈上。

注意：注意，scratchpad 在内存中的位置并非固定不变。它仅仅是字符串的临时存储之处。当有新的词被定义或存储空间被重新使用时，pad 所使用的地址将会有所变动。

举例来说，如下的程序可从控制台输入 20 个字符：

```
pad 20 expect
```

堆栈操作

因为所有的词在执行时都会使用堆栈上的参数，所以 Forth 提供了一些工具，以便程序员改变堆栈的内容，或是修改堆栈中值的顺序。

Forth 词 .s 可以在不移除任何单元的情况下，显示堆栈当前的内容。这是一个相当有用的工具，让程序员得以对堆栈进行非破坏性的监控。

其他有用的堆栈词包括 dup（复制栈顶的单元）、drop（丢弃栈顶的单元）、swap（互换栈顶的两个单元）以及 over（将“从栈顶数第二个单元”复制到栈顶）。

例如，键入“5 dup”会使堆栈出现“5 5”的结果。“3 4 swap”会使堆栈出现“4 3”的结果。“1 2 over”会使堆栈出现“1 2 1”的结果。dup 最常用于保留堆栈上的单元，以避免使用这些单元的“词”把它们从堆栈中移除。例如，下面的 Forth 程序将会打印出 A 这个 ASCII 字符相应的数值：

```
65 dup emit space . cr
```

因为 emit 与“.”都会从堆栈中取出一个参数，所必须用 dup 产生两个副本供两者使用。

?dup 只会在栈顶值不为零的状况下进行复制操作。而 2dup 则会同时复制栈顶的两个单元。例如：

```
10 23 2dup
```

会产生如下的结果：

```
10 23 10 23
```

同样，2swap、2over 和 2drop 也会同时操作栈顶的两个单元。

词 rot 会将（从栈顶算）第三个参数回转至栈顶。例如：

```
1 2 3 rot
```

会产生下列结果：

```
2 3 1
```

词 -rot 会从另外一个方向进行回转，效果则与 rot 的顺序恰好相反：

```
1 2 3 -rot
```

会产生下列结果：

```
3 1 2
```

词 pick 允许从堆栈取任何单元，其位置从栈顶单元开始算起。例如“. *n* pick”将会取得从栈顶单元算起的第 *n* 个项目。

警告：pick的操作方式取决于你所使用的Forth版本。相关细节请参考相关说明文件。

在Forth-79和FIG Forth中，pick的编号方式从1开始算起。所以如果我们的堆栈看起来如下所示：

```
54 46 32 29 10 5
```

则：

```
3 pick .
```

将会把29这个数输出到控制台上。

然而，在Forth-83、ANSI Forth和GNU Forth中，pick的编号方式从0开始算起。如果你所使用的Forth是这些版本中的一种（具有相同的堆栈值），则在上述堆栈下：

```
3 pick .
```

将会把32这个数输出到控制台上。

词nip会从堆栈上移除第二个项目，而词tuck会将最顶端的单元复制到第三个位置。例如：

```
78 12 nip
```

会得到如下的结果：

```
12
```

而：

```
78 12 tuck
```

会得到如下的结果：

```
12 78 12
```

创建新词

要创建一个新词，可以使用冒号定义（colon definition），示例如图3-1所示。这个简单的程序会将字符A输出到控制台上。

冒号(:)用来通知编译器，这是一个定义的开始，而接下来是新词的名称，此例为fred。最后，用分号（;）来结束此定义，这相当于C或汇编语言中的return。这个新词是

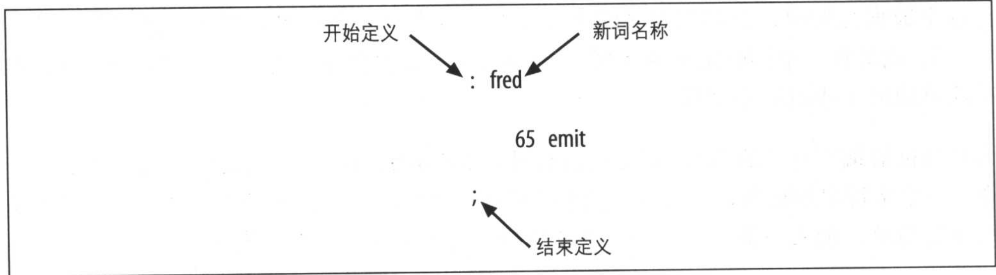


图 3-1：冒号定义

由冒号和分号之间的内容所构成的。此例中，十进制数 65（字母 A 的 ASCII 编码）会被压入堆栈，然后词 emit 会被调用。要执行这个新词，只需在命令提示符下键入 fred 即可。要执行 fred 三次，只需在命令提示符下重复输入 fred fred fred 即可。

Forth 使用线程码（threaded code），这是一个子例程标志符（字）清单。程序中所包含的每个词会被依次调用，因此会产生一组用来执行的程序码。由解释器负责适时地调用这些词，解释器只会进行三项操作。第一项操作是 *call* 操作，顾名思义，也就是开始执行所指定的词。第二项就是所谓的 *next* 操作，也就是将控制权从一个词传递到清单中的另外一个词。第三项是 *return* 操作，也就是将控制权返还给词的调用清单。call 操作数常被称作 *nesting*（嵌套），而 *return* 操作数常被称作 *unnesting*（解除嵌套）。由于采用这样的结构，所以 Forth 具备循环的能力；循环的层数仅受限于存储空间。

Forth 将会通过两种堆栈来完成此操作。第一种堆栈是参数堆栈，用来在词之间传递数据。第二种堆栈是返回堆栈，用来保存目前正在执行的词的返回地址。一般来说，返回堆栈就是处理器的系统堆栈。通常，这两种堆栈在 Forth 的虚拟机中得以实现（Java 的虚拟机也采用相同的方式）。此虚拟机还具有一个与传统处理器的指令指针（或程序计数器）等效的解释指针（Interpretive Pointer，IP），以及一个用来跟踪定义中各个词的词指针（Word Pointer，WP）。

Forth 的词会被存放在一个称为词典（dictionary）的链表中，如图 3-2 所示。

	名称	代码地址	下一词
1000	test	2000	1010
1010	fred	2010	1020
1020	check	2036	1030
1030	status	20F5	1040
1040	calc	2107	1050

图 3-2：词典的结构

在这个数据结构中，会按照定义的次序来存放各个词。所以当使用者在控制台上键入fred，或是在一个词的定义中使用fred时，Forth就会在词典中寻找相应的项目，然后简单地调用适当的子例程。

为目标机器编写自己的Forth系统也很容易。所需要的只是一组用来作堆栈的的基本命令、一个字符串解析器，以及一个简单的编译器（只需具备以下功能：建立一个子例程的地址清单、加入一条返回指令，以及随后在链表中加入一个新的项目）。

当程序员重新定义一个词时，新词在词典中会出现在旧词的前面。当Forth的解释器对输入进行解析时，就会在词典中寻找相应的项目。因此，一个重新定义的词会在解析器抵达旧词之前先被比对到。之前曾用到旧词的定义，由于它所引用的是该词的地址而不是其词典的项目，所以依然会指向旧词。因此，重新定义一个词将不会影响到基于旧词定义的其他词。然而，如果重新定义一个词是为了修改瑕疵，那么基于旧词定义的其他词则必须重新编译，以便使用较新的定义。

还可以根据旧定义的变换来重新定义一个新词。例如，词fred词可以被重新定义成执行它先前的定义两次：

```
: fred
    fred fred
;
```

此例中，新的词典项目fred指向用来执行fred的旧定义两次的代码，其所引用的是fred的原始定义的地址。当编译器为fred建立一个新的项目时，它会在词典中搜索旧的引用，并将一个子例程调用加到适当的位置。fred的原始程序仍在存储器中，而且会被其他词调用。当你在命令提示符下键入fred以便执行新词时，Forth会搜寻旧词典，直到发现相符的项目为止。Forth会先找到较新的定义，也就是所要执行的程序。那些使用fred之前定义的词不受影响，因为它所引用的是它在内存中的地址。

由于Forth本身就是用Forth编写的，而且编译器和语言本身就是词典中的词，所以你可以在编译器执行的时候重新定义编译器！在C语言中试着这么做看看！

注释

Forth的注释放在一对小括号里：

```
( Here is a comment )
```

注意，这对小括号的前后都有空格。因为这对小括号是Forth的词，因此需要加上空格，省略空格就会失去注释的作用：

```
(This ISN'T a comment)
```

Forth 将会试图把“(This”、“ISN’ T”、“a” 和 “comment)” 解释成个别的词，而且如果无法在词典（词清单）中找到它们，Forth 就会报错。

注释还可以用来说明堆栈图（stack diagram）。堆栈图指出了一个词的堆栈用法以及它的功能。堆栈图的一般形式指明要从堆栈取走的参数（N1）、要放回堆栈的参数（N2）以及该词的用途：

```
Wordname (N1 -- N2 , what this word does)
```

下面是一些 Forth 常用词的堆栈图：

```
1+ ( N -- N+1, increments the top of the stack )
dup ( N -- N N, duplications the top of the stack )
swap ( A B -- B A ,swap top two values on stack )
```

通常，我们会在定义一个词的时候使用堆栈图，以便说明堆栈的使用和该词的用途。下面是一些例子：

```
: helloworld ( -- , say hi )
     ." hello world"
;

: square ( A -- A*A , squares top of stack )
     dup *
;
```

因为堆栈图是注释，具体要如何指定参数实际上由你自己决定，只要对你以及其他打算阅读你程序的人有意义就可以了。

if…else

Forth像其他编程语言一样也支持if...else...语句，不过它的结果与你所熟悉的方式大相径庭。if 会将堆栈最顶端的单元当成一个布尔值。任何非零值都会被看作 true。在下面的例子中，你可以看到使用 if 的一些示例代码：

```
: is_it_true
     if
          ." That is true"
          cr
     else
          ." Not true
          Cr
     Then
;
```

注意，if语句终止于then。假如你已经习惯于其他编程语言的语法，就会感觉违反了常规。如果我们键入：

```
5 is_it_true
```

就会得到：

```
That is true
```

但是，如果我们键入：

```
0 is_it_true
```

则会得到：

```
Not true
```

一个简单的不需要使用else的if语句，其格式可简化成以下的样子：

```
: is_it_true
     if
           ."That is true"
           cr
     then
;
```

现在，如果不使用then来终止if结构，你还以使用endif：

```
: is_it_true
     if
           ." That is true
           Cr
     endif
;
```

then和endif的功能是一样的。使用endif将会让程序变得容易阅读。至于选择哪种结构，则由你自己来决定。

循环

你可以使用Forth词do和loop来建立循环。接下来的简单示例将会输出5个空白列：

```
5    0    do
     cr
loop
```

在C语言中等效的循环如下：

```
for ( i = 0; i < 5; i++ )
{
     printf("\n");
}
```

注意：在某些版本的Forth中，本节所提到的循环结构只能在词的定义之中使用。GNU的GForth便是这样。在其他版本的Forth中，可以直接从命令行来使用它们。这取决于你所使用的版本。

在C语言中，循环计数器不一定每次都要递增1。Forth也没有这样的限制。有此需要时，可以使用词+loop，它会从堆栈取一个值，用来递增循环计数器。在下面的例子中，循环每次都会递增2。

```
10 0  do
    cr
2   +loop
```

这非常有用。前面的例子中，循环每次所递增的值都被指定在+loop之前。然而，因为+loop要从堆栈取一个值，这个值可以等到执行的时候再指定，而且只要变更堆栈的单元，就可以在每次循环时使用不同的递增值。

do会从堆栈中取出两个操作数，作为循环计数使用。在前面的例子中，循环结构会执行词cr五次。你可以通过其他的词来提供这些操作数。下面的例子中，循环会执行两次：

```
100    98    -     0     do cr loop
```

此外，定义一个词时如果使用do...loop，你还可以把它设计成从堆栈取回循环的操作数。接下来的例子中，循环的执行次数留给使用者在执行时指定：

```
: starts ( N -- , prints N asterisks to the console )
     0 do
     42 emit
     loop
;
```

要输出10个星号，可以这么做：

```
10 starts
```

在循环里可以通过词i来取得循环的计数值。在下面的例子中，我们会打印出循环执行时的循环计数值：

```
10 0 do i . cr loop
```

两个循环可以嵌套在一起。接下来的例子会在控制台上输出50个空格：

```
10 0 do 5 0 do space loop loop
```

要取得外层循环的计数值可以通过词j：

```
10 0 do 5 0 do i . space j . loop cr loop
```

当然，可以使用的不止是两个嵌套在一起的循环：

```
10 0 do                    ( loop 1 )
   5 0 do                  ( loop 2 )
     i j                   ( counts of loop 2 and 1 )
      20 0 do              ( loop 3 )
         i do              ( count of loop 3 )
         4 0 do            ( loop 4 )
          10 0 do          ( loop 5 )
            i j do         ( count of loops 5 and 4 )
          loop
        loop
      loop
    loop
  loop
```

正如你所看到的那样，无论在哪一层循环，i都可以用于取得现行循环的计数值，而j可用于包含现行循环之外的（外层）循环的计数值。如果是三个被嵌套在一起的循环，可以使用k来取得最外层的计数值。

do...loop的结构相当有趣，do会把堆栈栈顶的两个参数放到返回堆栈，并设置目标地址，让loop分支得以跳回来。此目标地址就是紧跟在do之后的词。当loop执行时，它会递增计数器并与返回堆栈上的循环边界值进行比较。如果与这个值相等，则循环就会终止执行；否则会跳回do所设定的目标地址继续执行。

使用leave指令可以提前结束一个循环。例如，如果没有任何按键被按下的话，下面的循环会执行10次：

```
10 0 do
  key? If leave endif
loop
```

key?会检查控制台是否有一个输入字符，并将一个布尔值压入堆栈，以指示此情况。if将会移除该布尔值，并有条件地执行词leave。

可以使用词begin和until来建立有条件的循环。Until将会检查堆栈栈顶的布尔值。如果此值为真，则终止循环；否则，跳回begin之后的词继续执行。接下来的例子会从控制台读取一个字符，直到“A”被按下：

```
: wait_for_A
    begin
```

```
            key
            65 =
        until
    ;
```

等号运算符（=）会从堆栈栈顶取两个值（此例中，就是key所取得ASCII码值，以及65这个值）并加以比较。如果这两个值相等，它会把一个真值（-1）压入堆栈；否则，就把一个假值（0）压入堆栈。Forth的=运算符等效于C语言的==。

另外一个有条件的循环结构是begin *condition* while *code* repeat。每次循环都会执行begin和repeat之间的代码，不过要满足适当的条件才可以。begin和while之间的代码用于评估条件，从而决定执行是继续还是终止。例如，接下来所定义的词，只有在使用者从控制台上键入一个A时才会执行循环：

```
: while_A
    begin
        key 65 =
    while
        ." That was a A"
        cr
    repeat
;
```

无限循环可以使用begin...again来实现。如下的词一旦执行就不会终止：

```
: semper
    begin
        ." Once more through the loop, dear friends."
        cr
    again
;
```

此类循环对于嵌入式系统非常有用，因为其中的程序必须不断地反复执行。

警告： Forth-83标准并未实现begin...again，此时可以使用begin 1 while...repeat或begin...0 until作为代替。

Forth-79、ANSI-Forth和GNU Forth都实现了begin…again，所以，如果你使用的Forth属于上述版本之一，就没有什么可担心的。

数据结构

Forth在不同的词之间通过堆栈传递参数，这使得该语言威力强大。堆栈使编程语言可再入（re-entrant）并支持递归（recursion）。然而，堆栈的动态本质使其不适合用于全局数据。幸好，Forth不仅可以使用堆栈，还支持命名常量和变量。

常量的声明方式为，先将常量值压入堆栈，再在词constant之后键入常量名称。例如，接下来的程序声明了一个名称为kilobyte（代表1KB所拥有的字节的个数）的常量，它的值为1024：

```
1024 constant kilobyte
```

要在程序中使用常量很容易，只要使用常量名称就可以把相应的常量值放到堆栈中：

```
kilobyte
```

例如，下面所定义的词会从堆栈取得值为kilobyte的数字，将其转换成字节，并输出结果：

```
: kilobytes ( N -- , convert N to bytes )
     kilobytes * .
     ." bytes" cr
;
```

所以，如果键入：

```
32 kilobytes
```

就会在控制台上看到如下结果：

```
32768 bytes
```

声明常量的另外一种方式为，定义一个可以把常量压入堆栈的词：

```
: myconstant
     908612
;
```

这样做的效率不如使用词constant，但作用是一样的。

变量的声明方式类似于常量，但是它们的用法是截然不同的。变量名称并不会把一个值压入堆栈，而是把一个指向变量的指针压入堆栈。变量的声明方式为，先将变量的初始值压入堆栈，再在词variable之后键入变量名称：

```
0 variable my_var
```

指向变量的指针常与@（发音为*fetch*）和!（发音为*store*）并用。假设我们要设置变量my_var的值为32，其做法是先将变量的值（32）压入堆栈，接着使用变量名称把指针压入堆栈，再使用!对变量赋值：

```
32 my_var !
```

要回调变量的值时，使用变量名称将指针压入堆栈，然后使用@取回该值并把它放到堆栈上：

```
my_var @
```

例如，假设我们要将 my_var 的当前值加上 5：

```
my_var @      ( get the current value )
5 +                ( add 5 )
my_var !           ( put the result back into my_var )
```

这看起来似乎有些奇怪，但是非常接近 C 语言中“my_var = my_var + 5;”语句背后操作的方式。在 C 语言中，你无法看到这个过程，而在 Forth 中，这些细节完全在你的掌控之下。

与硬件和存储器的交互

Forth非常适合用来调试嵌入式硬件。可以使用Forth来检查存储器或外部设备寄存器中的内容：

```
2000 @
```

这会把地址 2000 的内容压入堆栈。还可以使用 Forth 来设定存储器或外部设备寄存器的内容：

```
41 2000 !
```

该语句把地址 2000 的内容设定为 41。可以在一个词的定义中，或是在命令提示符下以交互的方式进行此操作。这意味着，你能够以交互的方式探测和改变外部设备的寄存器，这的确是一个非常强大的调试工具。

警告：如果在你的台式机（比如 Mac 或 PC）上使用 Forth 的版本，请勿轻易直接以 ! 或任何其他“词”来变更存储器的内容，因为这将造成意外（甚至是）灾难性的后果。

@ 和 ! 用来操作 16 位的地址。要存取 8 位的地址，可分别使用 c@（读作 c-fetch）和 c!（读作 c-store）。要存取 32 位的地址，可使用 2@ 和 2!。

警告：在 Forth 的某些版本中，比如 PowerPC 处理器上执行的 gforth，@ 和 ! 用来操作 32 位（而不是 16 位）的值。同样，在相同的平台上，2@ 和 2! 所操作的是 64 位的值。

例如，假设我们要建立一个名为 status 的 Forth 程序，它将会读取地址 2100 上一个 8 位寄存器的内容，并把结果压入堆栈：

```
: status 2100 c@ ;
```

这让我们得以在调试期间在提示符下手动检查寄存器的状态：

```
status .
```

当调试某些硬件时，或许你会持续监控状态寄存器。此时，可以利用下面的Forth程序，它会持续执行直到你在控制台上按下一个键为止：

```
: showstatus
    begin
        status .
        cr
        key?
    until
;
```

一旦调试完成，词status就可作为主程序的一部分：

```
: main
    begin
        status 5 =
    until
    41 TxRegister c!
;
```

词dump可用来探测一段存储器空间的内容。dump的执行方式为*address size* dump，所以需要从堆栈取得两个操作数。例如，要从scratchpad当前的位置开始输出256个值，可以这么做：

```
pad 256 dump
  BB1D4: 0C 30 C3 0C  C3 0C 30 C3 - 0C 30 C3 0C  33 33 33 30  .0....0..0..3330
  BB1E4: C3 0C 30 C3  0C 30 C3 0C - 30 C3 0C 30  C3 0C 30 30  ..0..0..0..0..00
  BB1F4: C0 C0 C0 C3  0C 2C 2F 89 - 09 24 24 90  92 49 09 02  .....,/..$$..II..
  BB204: 42 00 00 00  20 B0 B0 B0 - 90 90 B0 00  3C 00 00 00  B... .......<...
  BB214: 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00  ................
  BB224: 00 00 00 00  00 00 00 00 - 00 00 22 42  42 42 49 32  .........."BBBI2
  BB234: 4C 92 4C 24  09 02 42 42 - 40 92 49 09  09 24 24 90  L.L$..BB@.I..$$.
  BB244: 24 09 02 42  4B C0 BF CB - C9 30 AD C3  0B 30 2F F0  $..BK....0...0/.
  BB254: 0B B8 BF 82  FE 2B 8B E2 - FC BF 2F E2  FF FE 2F FC  .....+..../.../.
  BB264: 2F 8B FF BF  2F FE FC 2B - 7F BB DD FF  7F F8 2F 7D  /.../..+....../}
  BB274: F8 AF FF BF  E2 F0 B7 FC - AC 2A D8 AB  FF F7 F0 AD  ........*......
  BB284: E2 FE DF 0A  F0 B7 DE FB - C2 DF E0 AF  2F FE 2F CB  ...........././.
  BB294: E2 AE 2A E0  90 90 BF 0B - F0 BE 2F 8B  2C B2 CB F8  ..*......./.,...
  BB2A4: BE 24 92 49  24 92 4B FF - 2F 8B F0 80  00 02 DB F7  .$.I$.K./.......
  BB2B4: FE 08 00 00  2D BF 7F E2 - FF 2B FF BF  2F FE 2F FE  ....-....+../../.
  BB2C4: 2F FF F8 BF  DF FF 8B DE - 2B EF B7 CB  FE 2F FF FF  /.......+..../..
```

对所输出的每一行，dump都会输出一个跟有16字节数据的地址，以及这些字节所对应的ASCII码。在这些十六进制的字节数据中，对于无法对应到可打印ASCII码的字符，则会打印出点号。

要清除存储器空间（也就是把每个地址的内容置为0）可以使用词 erase。例如，要将 32 个零写入从地址 2100 开始的位置，可以这样做：

```
2100 32 erase
```

与erase类似，fill可用来把一个特定的值填充到一段存储器空间中。例如，要将255 这个数填入从 2100 开始的位置，可以这么做：

```
2100 32 255 fill
```

Forth 程序设计准则

如同其他程序设计语言一样，要写出一个优秀的 Forth 程序并非不可能，但是要写出一个糟糕的 Forth 程序也不是什么难事。下面是一些 Forth 程序设计的准则：

- 堆栈的使用限制在三到四个词之内。如果超出此范围，你会发现很难记住自己的程序做到哪里了。
- 词的定义不易太长。千万不要定义出一个含有上百行程序的词。Forth 的优点在于该语言能构建其自身，使你做完后可以立即测试它。为此，请尽量将每个词的程序长度维持在三到四行范围内。如果你习惯于其他编程语言的话，这听起来可能有点奇怪，不过一旦你熟悉 Forth 的节奏之后将会发现，这其实是一个符合逻辑的编程方式。
- 词的定义要尽量简单，并把重点放在你想要达成的目的上。一个只能解决特定简单问题的词，胜过一个复杂而什么都能做的词，因为这样的词既难写又难调试。
- 检查数据是否超出范围或者不正确，并产生有用的错误信息或调试信息。

Forth 的威力远不止这些。更多信息可以访问以下网站：Forth 兴趣小组（Forth Interest Group，FIG）的 *http://www.forth.org* 或 Forth ,Inc. 的 *http://www.forth.com*，以及 Forth 的发明人 Charles Moore 的个人网站 *http://www.colorforth.com* 获得。如果你对 Forth 有兴趣，互联网上有许多免费的版本供下载。比如，GNU 版的 Forth 可到 *http://www.gnu.org/software/gforth* 下载。当然，也可以通过搜索引擎寻找适合自己的 Forth 版本。

第4章

电子学概览

……在物质世界里，除了原子和真空就别无他物了。

——德谟克利特

在撰写本书的过程中，我一直怀着这样一个希望：让你了解构建一台嵌入式计算机系统的设计过程。为此，我尽可能简单地描述我所涉及到的电子设备、芯片和系统。我想让你了解设计嵌入式计算机的概况，而不是迷失于设计的具体细节中。但是，无论我如何尽可能地简化嵌入式计算机的设计，你都不应该缺乏对电子学知识最起码的了解。因此本章提供了学习嵌入式计算机设计的基本背景理论 —— 电子学。电子学的确是一门广博、综合的多学科性专业领域，在一章里即便是想讲述其千分之一都不可能。我将做的是 —— 为你提供容易理解嵌入式计算机工程所必需的电子学基础理论。至于电子学的其他知识这里就不提及了。如果你想进一步学习，可以买一本由剑桥大学出版社出版的由 Paul Horowitz 和 Winfield Hill 合著的《The Art of Electronics》，那是一本很棒的入门级教科书。

电压和电流

电子学（electronics）是关于电子（electrons）的学科，所以叫作“电子学”。电子是带负电荷的亚原子微粒，由于库仑引力，电子被束缚在带正电的原子核的周围。在经典物理学看来，电子是围绕着原子核“旋转”的，类似于行星围绕着太阳系轨道运行。这种观点虽算不上完全正确（注 1），但形象地描述了原子内电子和原子核的运行情况，便于

注 1：　事实也总是奇异得很。量子论的观点是美丽而奇妙的。如果想获知有关量子论的简单的、极好的介绍，就请阅读 Richard Feymans 的著作《Quantum Electro Dynamics》（普林斯顿大学出版社出版）。

人们理解。电子受到的原子核的力随着原子的不同而不同，也随着分子的不同亦有不同。所有的物质，要么是导体、绝缘体，要么就是半导体。在导体里，如金属，将一个电子从一个原子核移动到另一个原子核所需的能量几乎可以忽略，因为电子可以轻易地在相邻的原子核间交换位置。实际上，金属是一个由“大量”半自由的电子围绕着的原子核集合。在绝缘体里，情况恰恰相反，需要非常大的能量才能将一个电子从其原子核移开，这种电子倾向于固定不动。在半导体里，物质既可以是导体，也可以是绝缘体，具体取决于外部的影响。通过控制外部作用，就可以改变半导体物质的传导性，从而改变这类物质内电子的运动方式。实际上，一个半导体就是一个开关，而这一开关可以由其他半导体来控制。这一基本原理是所有现代电子学的基础，也是任何事物数字化的基石。

通过导体或半导体的电子流，就是所谓的电流。电流是以安培（Amperes）来度量的，通常简称为安（Amps，单位符号为A）。要使一个电子通过导体（注2），就必须在下一个原子核的附近位置有个“空缺”，以便移进这一电子（如果下一个原子核附近充满了电子，那么这些电子的库仑斥力将阻止任何其他电子的移入）。半导体物理学家将上述空缺称为“空穴（holes）”。一个电子转移到一个邻近的空穴里，而其后又留下一个新的空穴。而这一新的空穴又被其下另外一个电子所填充，这样就建立了另外一个新的空穴。因此，所谓的电流实际上是电子在一个方向上的运动，同时也是另外一个方向上的“空穴运动”。电子是带负电的，而上述“空穴”就可以看作是带正电（一个原子核附近消失一个电子意味着该原子核的正电荷没有完全被取消，因此在电子消失的位置就存在一个净正电荷）。因此，当电子从负极移到正极的同时，空穴也从正极移到负极，我们所指的电流正是空穴（而不是电子）的移动。电子学中的电流设定为是从正极到负极流动。为了得到持续的电流，就必须有电子在一个方向上的循环流动和空穴在另外一个方向上的循环流动。也就是从循环流动中，得到“电路”这一术语。

两点之间要出现电流，那么一端的电子与另外一端的空穴之间必将存在一个不平衡。这一不平衡的大小就是两点间的电势差或电压差（这有时也称作“电压通过一个电子元件时的电压降”）。电压差的基本单位是伏特（Volt，单位符号为V）。电压差越大，电流发生的机会就越大。应该着重指出的是，电压是指两点之间的电势差值。电压不存在于单独的端点中。虽然有时会看到像“该点的电压是多少多少伏”这样的陈述，但这个电压值是相对于某一公共参考点而给出的，通常是接地点（即零电压参考点）。

警告： 在测试电子电路时，初学者容易犯的一个普遍错误是只测量待测设备一端的电压。实际上，如果没有测其两端的电压，也就没有公共参考点存在；进而所有的测试都是毫无意义的。

注2： 虽然半导体是正规的导体，但我还是把它视作传导半导体。

模拟信号

一个模拟信号可以拥有一定范围内任何电压的幅值，而不像数字信号那样只能处于两种定义电压状态（要么低电平，要么高电平）之一。图4-1所示为一个典型的模拟信号（本例为正弦信号）。

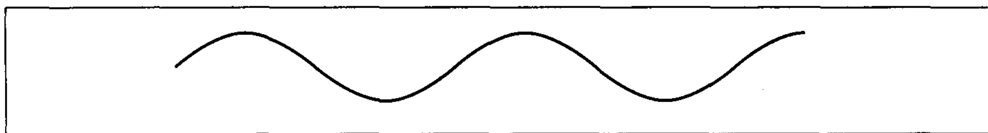


图 4-1：模拟信号波形

信号的电压可以随着时间的推移而变化，或者为常值。如果电压是变化的，那么它会每隔一定时间就重复，这种情况下就说这种信号具有周期性。这个周期就是信号模式重复的时间间隔（例如，从一个波峰到另外一个波峰）。信号的频率就是每秒钟信号模式重复的次数。

频率的单位是赫兹（单位符号 Hz），它与周期有如下关系：

```
f = 1 /T      (f: 频率; T: 周期)
```

这样，周期为 1ms 的信号，其频率就是 1kHz。

一个单极信号电压要么全为正要么全为负，如图 4-2 所示。一个双极信号具有正和负两种电压，如图 4-3 所示。

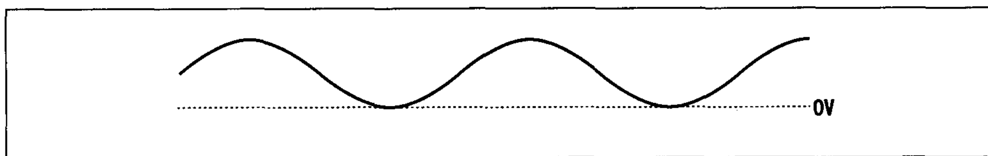


图 4-2：单极信号

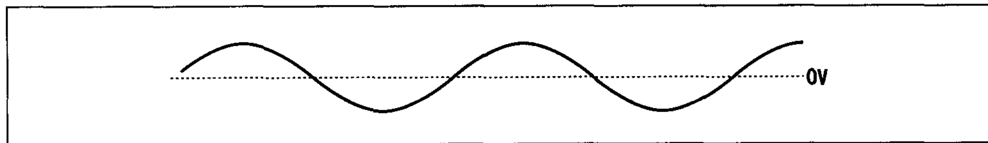


图 4-3：双极信号

典型的模拟信号既有交流（Alternating Current，AC）分量又有直流（Direct Current，DC）分量，如图4-4所示。DC分量是信号的固定电压，而AC分量则是一个加在DC分量上的变化着的电压。AC分量通常是指信号的峰－峰振幅，其后一般以后缀pp来表示。例如，一个5V的交流分量可以写作5Vpp。

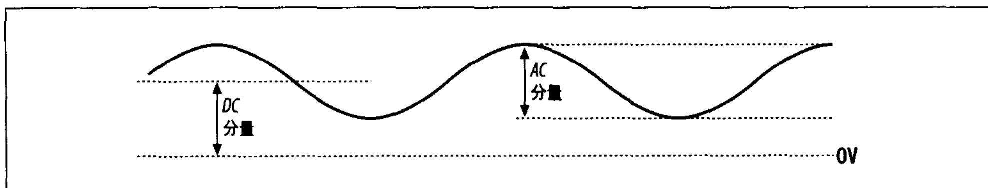


图4-4：模拟信号的直流和交流分量

功率

电压是由两点之间的势能差产生的。因此，要产生一个电压，就得使用能够产生一个能量差的设备。这样的一种设备可能是机械式的（发电机），它通过电磁、光电（太阳能电池）或者化学制品（电池）来把运动能转化成势能。相反地，势能（继而电流）可用来产生机械运动（电动机）、光发射（电灯泡、LED）以及热量（烘炉、Pentium 4处理器发热）。

功率是单位时间内所做的功（焦耳/秒），单位是瓦特（单位符号W）。计算功率的公式很简单：

```
P = V * I      (V: 电压; I: 电流)
```

任何电子设备的效率都不可能是100%的（与此相差甚远），因此在工作时它将消耗能量。设备的功耗用上述公式可以计算出来，即利用设备的电压差和通过该设备的电流来计算。一个典型的嵌入式计算机可以消耗几百毫瓦（mW）的功率，但不同的嵌入式机器的功耗可能相差很大。一个大型的强大嵌入式机器可能要用几十（甚至上百）瓦，而一个小巧的嵌入式控制器可能仅需微瓦（μW）的功率（$1\mu W=10^{-6}$ W）。

理解电路原理图

除非你知道如何画出或读懂电路原理图，否则你不可能深入学习电子学。电路原理图无处不在，理解它们是必需的。电路原理图就像建筑师的设计图。图中表明电路里所用到的元件以及这些元件的连接方式，也可能包含诸如设计指导之类的信息。电路原理图里还可能有一个修正列表，表明自最初设计以来电路原理图都有哪些改变。这些内容叫作

工程变更清单（Engineering Change Orders，ECO）。随着设计的提高和改变，对所做的修改以及为什么要做这些修改进行跟踪是个很好的想法。正如给源代码加注释一样，记录ECO也很重要。

你将遇到两种类型的电路原理图。你会在数据手册里看到电路原理图，在书里（比如本书）和其他技术文档里也可以看到，这种电路原理图只会给出电路（或部分电路）以及一个或两个注释，仅此而已。另外一种类型是实际用来制电路板的原理图，它们描述了一个完整的系统设计，并且一般会在图纸右下角有个标题框，表明这幅图纸的用途以及它的设计工程师和设计时间。图4-5给出了这样一个标题框示例。

Title	HAL 9000		
Size A2	Number 920112-890		Revision A
Date	11-Jun-2002		Sheet 1 of 51
Drawn By	Dr Chandra	Checked By	

图4-5：电路原理图标题框

从本质上讲，在一个电路原理图上有两类对象：元件和网路（net）。网路表明元件之间是如何相连的。元件有元件名和元件类型。例如，一个存储器芯片会有元件名U3和元件类型AT45DB161。元件名就像源代码中的变量名，只是一个简单的索引标签，而元件类型则是生产厂商使用的一个现行器件号码。

比较常用的做法是对同类型的元件名冠以同一前缀。例如，电阻的前缀为R，你会看到电路原理图上的电阻标为R1、R2、R3等。类似地，电容的前缀为C，电感的前缀为L，二极管为D，晶体管为Q，晶体为X，而连接器和跳线器为J。半导体的前缀通常为U，但也不是绝对，逻辑门和其他难以区别的小半导体可能也有前缀U，但大的半导体一般会有更具提示性的元件名。例如，一个处理器可能标为PROC，而四个存储芯片会标为RAM0、RAM1、RAM2和RAM3。为大一些的设备提供更富意义的名字常常有助于理解电路原理图。然而，仍然有很多人对所有的半导体只标前缀U。

图4-6所示为一个由网路和元件组成的电路原理图的例子。

除了名字和器件号码，元件还有一系列引脚。引脚也有编号或名称，或两者都有。编号表明了原理图引脚所指的芯片上的物理引脚，名称表明了引脚的功能。有些器件，如电阻，既没有引脚名也没有引脚编号。

元件的引脚可能会有名字和标号，以指明元件的特征。图4-7所示为带有各种引脚的一个元件。

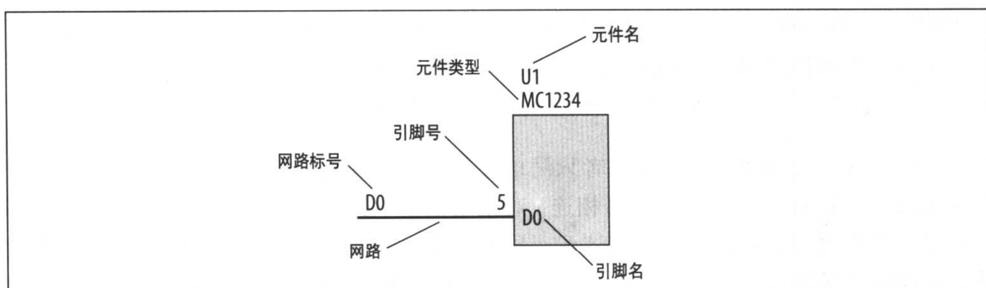


图 4-6：信号线路和器件

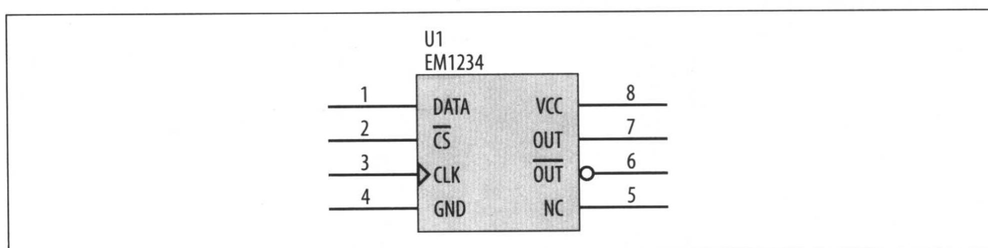


图 4-7：引脚类型

引脚1是个普通引脚，引脚2在引脚名字上方有个上划线，表明这个引脚是低电平有效，这意味着在该引脚上的逻辑0会激活它的功能，而逻辑1会解除它的功能。引脚2的名字为$\overline{\textbf{CS}}$，典型含义是片选，也就是说这个引脚用于激活芯片。大多数外部设备和存储器芯片都有一个片选输入。片选很重要，因为计算机系统内有很多存储器和外部设备。正是通过片选，处理器才能够激活这一芯片，从而可以向其写入数据或者从其读出数据。有些设备有一个输入引脚叫作$\overline{\textbf{CE}}$，即芯片使能（Chip Enable），它几乎完全等同于片选引脚，不过是同一功能引脚的两种不同叫法。

引脚3上的小三角表明该引脚是一个边缘触发输入。

引脚4和引脚8分别是接地（GND）和电源（VCC）。VCC和VDD用来标明给电路供电的电压源。该术语源自晶体管和固态电子设备，其中“集电极（VCC）”和“漏极（VDD）”都是常用的说法。你不必考虑名称的具体含义，只需看到VCC或VDD时知道该引脚是与供电有关的即可。

引脚7是一个高电平有效的输出引脚，引脚6也是一个输出引脚，但低电平有效。注意引脚6上有个小圆圈，表示它是一个反相输出。引脚6和引脚7有同样的名字，但引脚6是引脚7的输出的反相。最后，引脚5标为NC（No Connect），这通常代表“没有连接”，表示该引脚没有功能，即没有元件与该引脚相连（你可能很少见到引脚名为

“DNW”，即 Do Not Wire，其意义与 NC 一样）。然而，看到一个标为 NC 的引脚也不一定就意味着可以认为在此点不会有连接。有可能只是芯片制造商为了其他原因而将其标为 NC。总而言之，你最好查看每个设备的技术手册。

可以在两个元件之间进行连接，或者只简单地用网路名来给出网路标号（net label），表明它跟每一个具有同样名字的网路相连。对于复杂的电路，把每一处连线都画出来也不太现实，因为这可能弄得到处都是连线，导致电路原理图难以理解。因此，通常的做法是简单地使用网路标号来在局部命名一个网路，仅此就可以表明哪些元件之间彼此相连，如图 4-8 所示。

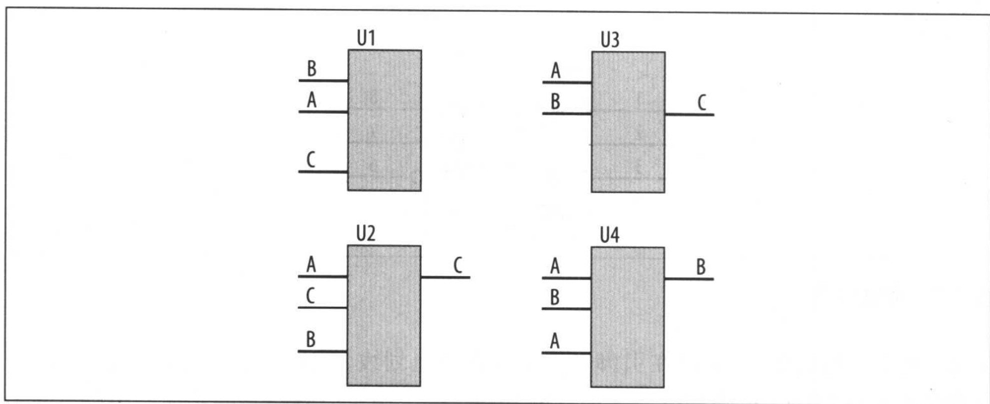


图 4-8：无需连线就可表明元件连接的网络标号

功能相关的信号，如总线，就是使用总线网路（bus net）来画的，如图 4-9 所示。

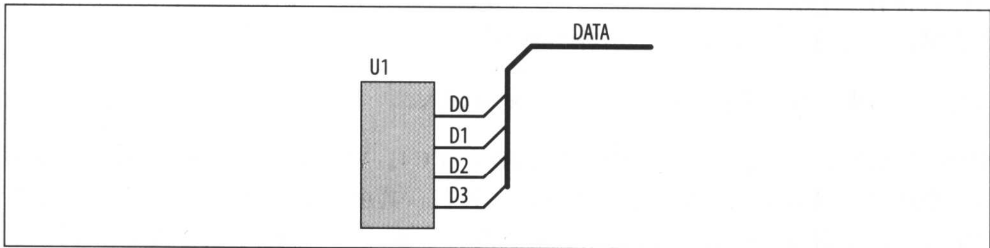


图 4-9：相关信号用总线来发送

一个电路设计经常会采用不止一个电路原理图。正如把程序分成函数，把经常使用的代码放到库里一样，在进行电路设计时也可以把整个设计划分为若干功能单元，以便这些子系统在多个设计中得以重复使用。例如，同样的电源电路可以在几个不同的嵌入式计

算机设计中使用。通过把电源电路放在单独的原理图上，同样的子系统设计就可以在许多设计中复用。当电路原理图上的一个网路连接到另外一个电路原理图时，就使用端口（port）来指明。

图4-10给出了一个连接于电路原理图之外对象的元件。在本例中，D0:D3端口是双向总线，A0:A3端口对这个电路原理图而言是输入总线（对另外一个电路原理图而言就是输出），而MODE端口对这个电路原理图来说是输入网路。

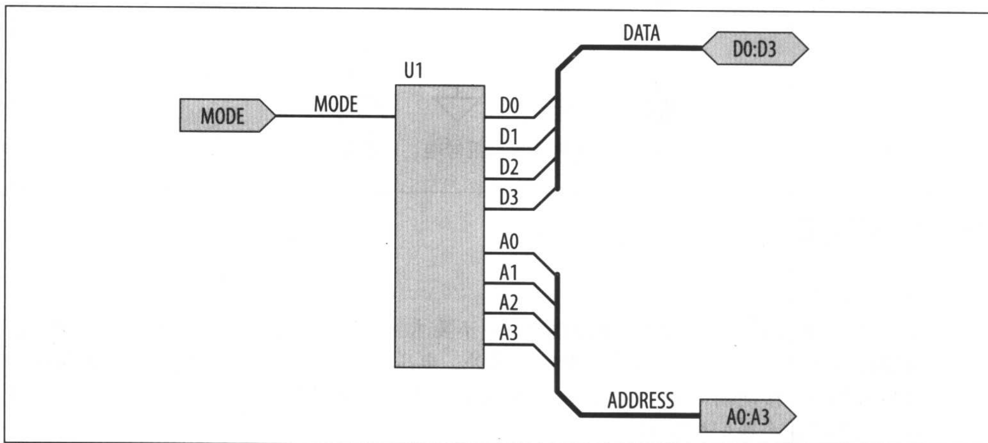


图4-10：端口表明网路对多个电路原理图进行连接

图4-11给出了相互交叉的网路。左边那个垂直的网路没有连到水平的网路，它反连到网路里的另外一个部分。右边那个垂直的网路连到了水平的网路，这通过交叉点来表明。

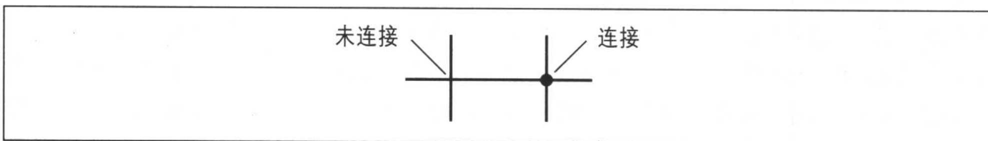


图4-11：网路交叉

在一些计算机爱好者杂志和比较旧的课本里，你有时会看到一个网路穿过其他网路时使用了桥接符号，如图4-12所示。这绝对不是画电路原理图的方法——很不专业，很不老道。

图4-13给出了常见的电源端口。这些端口表明它们是连接到电压源（供电）还是接地。接地符号意味着零电压。不同的符号用以区分不同的接地网络。在微处理器电路原理图里，经常会见到最左边的两种接地符号（通常为两者其一），而很少看到另外两种。

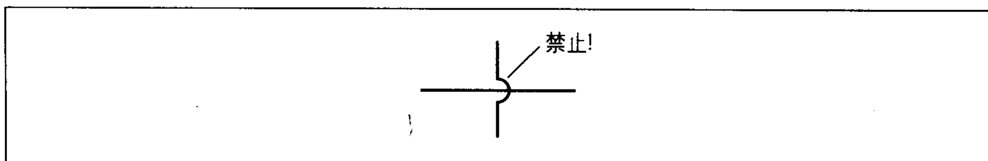


图 4-12：不要这样画交叉网路

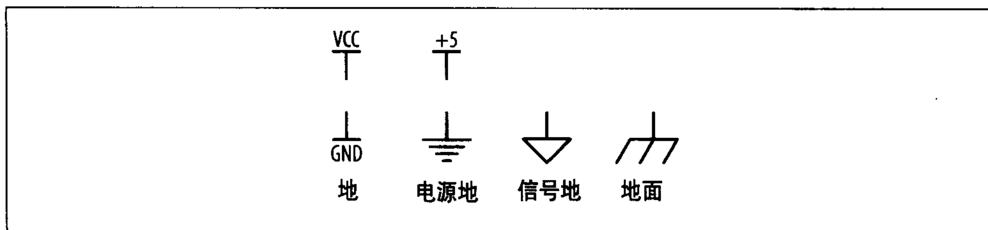


图 4-13：电源端口

注意： 顺便说一句，请垂直放置你的电源端口，而不要水平放置。水平放置的电源端口就如同没有缩排的代码 —— 这种无知令人蹙额。另外，电压端口（如 VCC）应该朝上，而接地端口应该朝下。接地端口绝不应该朝上，电源端口也不应该向下。对于专业工程师而言，这种不当的做法令他们烦恼不堪。

电阻

即使导体（如金属线）在传输电流时也不是 100% 有效率的。电流通过导线，能量将以热能的方式（有时为光）消耗掉。对于非常小的电流，这种能量损耗是微不足道的，但对于大的电流，这种损耗可能导致导线过热（一个有效的利用便是烤箱）或者光线极亮（灯泡）。能量的消耗导致跨越导线（或部件）的电压差，或者说该部件抵制电流。这种电阻（有时候也叫阻抗，虽然阻抗的含义较简单电阻更复杂）用欧姆（Ohms，单位符号 Ω，公式符号 R）来计量。电路原理图里一般省略 Ω 符号，因此 100kΩ 通常写作 100k。

注意： 在电路原理图上，一个 4.7kΩ 的值可能不是写作 4.7k，而是 4 k 7。这样做的原因是，当对电路原理图文件进行影印时小数点很容易被漏掉。解决的办法就是在小数点的位置放上乘数（k），如 24.9Ω 的电阻就写作 24R9。

世界上大多数设计工程师都采用这一惯例。然而，在北美洲依然采用原有的格式，这种表示方式只是偶尔使用。

电压、电流以及电阻之间的关系就是欧姆定律，由以下公式给出：

```
V = I * R
```

对于固定的电阻，变化的电压将产生变化的电流，而固定的电压也就产生固定的电流。因此变压电源叫作交流电源（AC），而恒压电源叫作直流电源（DC）。AC电压通常表示为VAC，而DC电压则表示为VDC，或更多地表示为V。

注意： 如果你生活在北美，那你墙壁上的插座供应的是交流电，为110~120VAC（频率为60Hz）；如果是在日本（在东京的东半部是50Hz，在西半部的大阪、京都、名古屋是60Hz）则是100VAC；而如果是在澳大利亚、新西兰、中国的香港或欧洲，则是220~240VAC（50Hz）。所有数字电子设备（包括计算机）内部使用的都是DC，工作电压通常是5V或者3.3V（有些数字设备工作电压为1.8V甚至更低）。计算机（TV、立体声系统或者其他系统）的供电设备把高压AC电源转变为电子设备所需的低压DC电源。AC电源适配器或者你的蜂窝电话上的插头（充电器）都是供电设备的例子。

对于给定的电压，电阻越小则电流越大。相反，电阻越大电流就越小。因此，电阻可以用来限定通过电路特定部分的电流。一种元件，即电阻，正是用于此目的。电阻是称作无源元件（passive component）的系列元器件之一。

直插式电阻在电路图里的符号如图4-14所示，从上到下分别是0.5W、0.25W和0.125W。

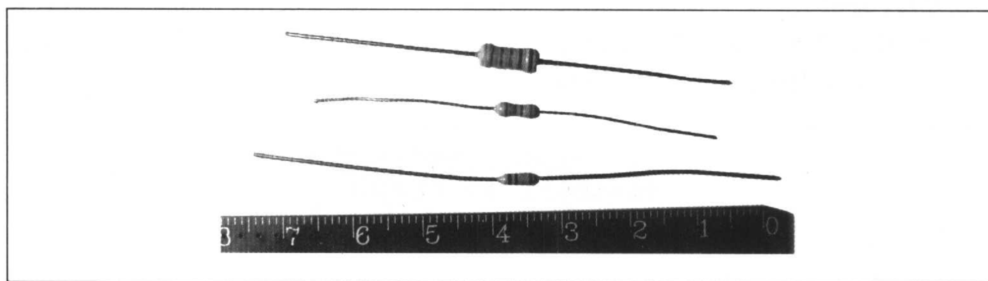


图4-14：直插式电阻（尺度为厘米）

图4-15所示为贴片电阻，左边是1206封装，右边是0603封装。目前贴片电阻技术已远比直插式电阻元件更普及。

还有更小封装的电阻，但如果你是手工进行焊接，可能会非常担心该元件在焊接到电路板上之前已经被毁掉了！

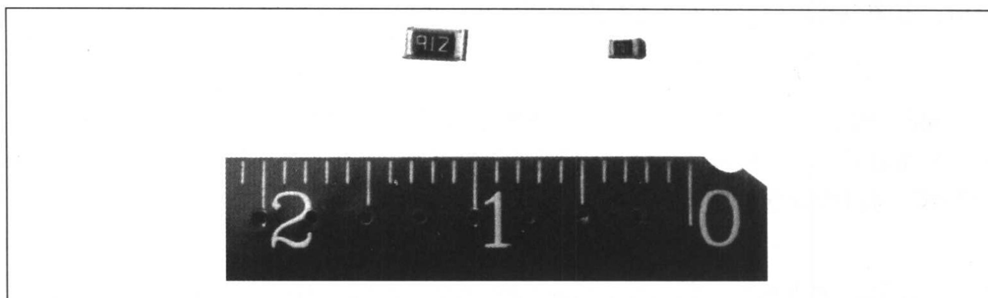


图 4-15：贴片电阻（尺度为厘米）

图4-16所示为电阻在电路原理图中的元件符号。这两个符号均代表电阻，不过左边那个更为常用。



图 4-16：电阻符号

电阻可以用来将一条信号线降（或升）至给定的值。如果你的电路中有许多上拉电阻，而且它们具有相同的阻值，你就可以用图 4-17 所示的排阻（resistor network），这样会比较方便。排阻是由多个电阻组合而成（通常为8个，但也可以有其他规模），并被封装到单一元件中。排阻中的所有电阻都有一端会被连接到一个引脚上，此引脚会被连接到电源，于是这些电阻就可以作为上拉电阻。

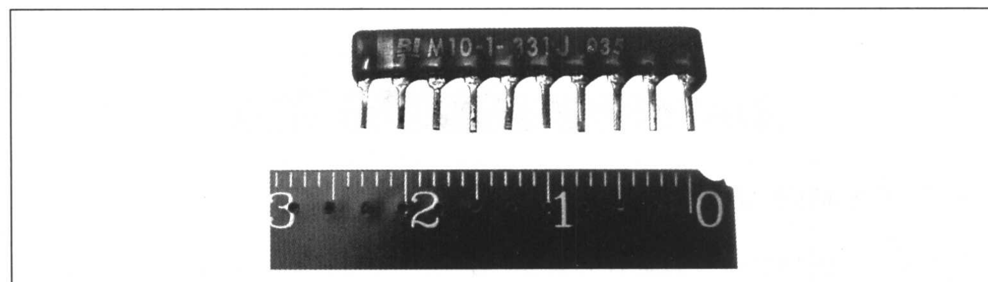


图 4-17：排阻

图 4-18 所示为一个上拉电阻和一个按钮。当按钮开（没有按下）时，电阻里没有电流通过，因此 V_{OUT} 的电压是（本例中）+5V（由于没有电流通过电阻，因此也没有压降产

生）。而当这个按钮按下时，V_{OUT}接地，电流就能通过电阻。这个简单的电路可以用于将输入在两个逻辑门之间切换。

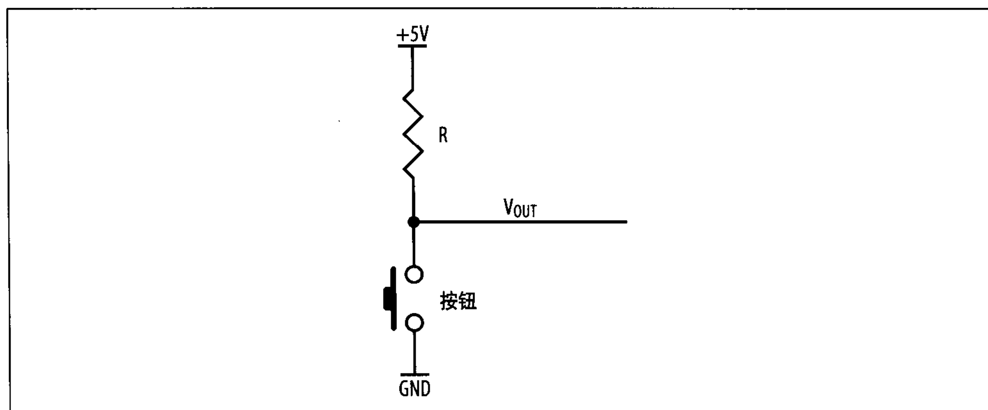


图 4-18：上拉电阻和按钮

电阻也可以连在一起来增加阻抗。图 4-19 所示是电阻的串联。



图 4-19：串联电阻

串联后的总电阻为：

$$R_{TOTAL}=R1+R2$$

在串联电路里，通过每个元件的电流都是一样的。换句话说，通过图 4-19 中第一个电阻和第二个电阻的电流都是一样的。这个结论由基尔霍夫电流定律可以得出。

注意：基尔霍夫电流定律：

电路中任意一个节点上的电流，等于流入该节点的电流之和，也等于流出该节点的电流之和。

换句话说，电流怎么流入就必须怎么流出。

电阻也可以用在分压器中来提供中间电压，如图 4-20 所示。

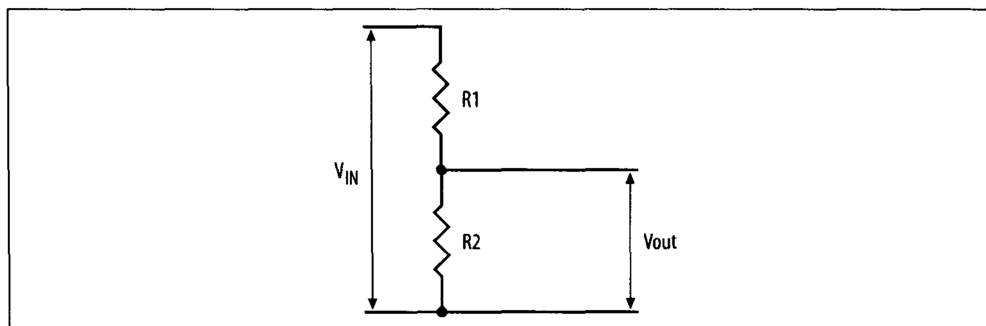


图 4-20：分压器

图 4-20 中的输出电压是：

$$V_{OUT} = V_{IN} * R2 / (R1 + R2)$$

例如，如果输入电压是 5V，两个电阻均为 1kΩ，那么输出电压为：

$$\begin{aligned} V_{OUT} &= 5V * 1k /(1k + 1k) \\ &= 5V * 1k / 2k \\ &= 5V * 0.5 \\ &= 2.5V \end{aligned}$$

正如你所期望的那样，使用两个相同电阻的分压器可以将输入电压减半。

并联电阻可以降低整个阻抗，如图 4-21 所示。

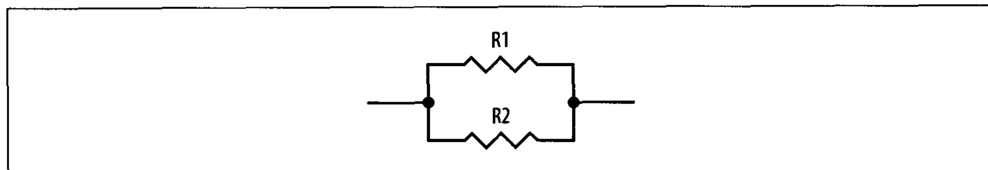


图 4-21：并联电阻

这种连接的总电阻关系是：

$$R_{TOTAL} = 1 / (1/R1 + 1/R2)$$

R1 和 R2 两端的电压一定是相同的，然而，除非 R1 与 R2 的阻值相同（没有必要一定相同），否则流过每一个电阻的电流是不同的。该结论可由基尔霍夫电压定律得出。

注意：基尔霍夫电压定律为：

闭合电路的电压差的和为零。

电位计（potentionmeter，有时简化为pot、trimmer或trim pot）其实就是一个可变电阻器，其电路符号如图4-22所示。电位计通常是机械元件，可以手动调整。在立体式音响的音量、低音和高音的控制中可能会用到它。监视器和LCD的亮度和对比度的调节中也可能会用到它。

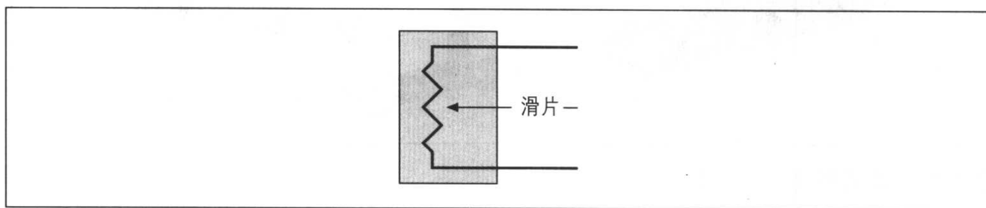


图4-22：电位计

一个标准的电位计包括连接到电阻的两端（图4-22中的上端引脚和下端引脚）以及一个称之为滑片（wiper）的第三端。滑片可以将电阻上下移动，以及从给定的位置有效地取出电压。当滑片朝一个方向移动时，可从滑片看出电阻值增加；滑片朝另外一个方向移动时，电阻值就可以减少。

电位计有多种不同的形式。图4-23所示为一种拉杆式电位计（slider pot）以及两种旋转式电位计（rotary pot）。通常，电位计会安装在装置内部，让滑片轴从面板适当的地方穿出，然后用螺丝将按钮固定在滑片轴上。

机械式电位计有各种阻值范围，不过其精度不高。此类电位计可用于调节电压输出（如图4-24所示）或是在模拟电路中简单地作为一个可变电阻。

举个简单的例子，你可以通过用电位计来改变流经LED的电流从而改变其亮度，如图4-25所示。其中，LED的阳极和电位计的滑片之间接有阻值为300Ω的固定电阻。通过调整滑片可以增加限流电阻的值，这样会减少通过LED的电流，进而降低它的亮度。

请注意，图4-25所示的电位计有一端没有连接。这很正常，因为在此例中并没有用电位计来提供某电压区间的中间值。我们只想增加滑片与VDD之间的阻抗，所以它被当成可变电阻器来使用。

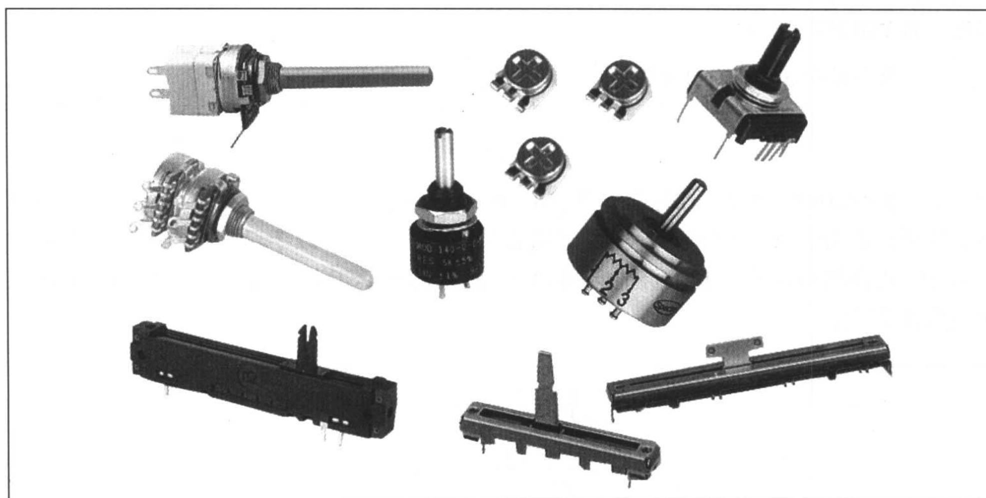


图 4-23：不同形式的可用电位计

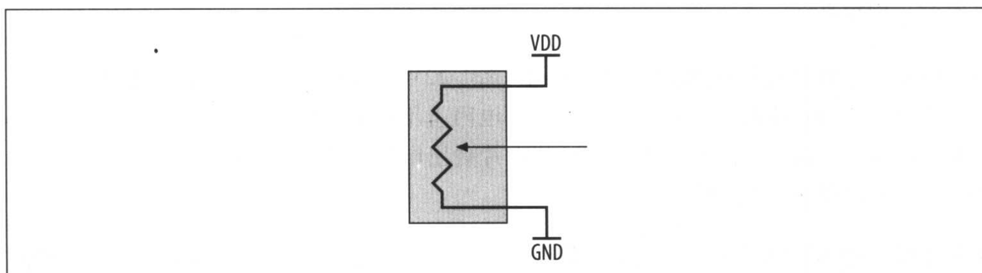


图 4-24：使用电位计在 VDD 与地之间提供可变电压

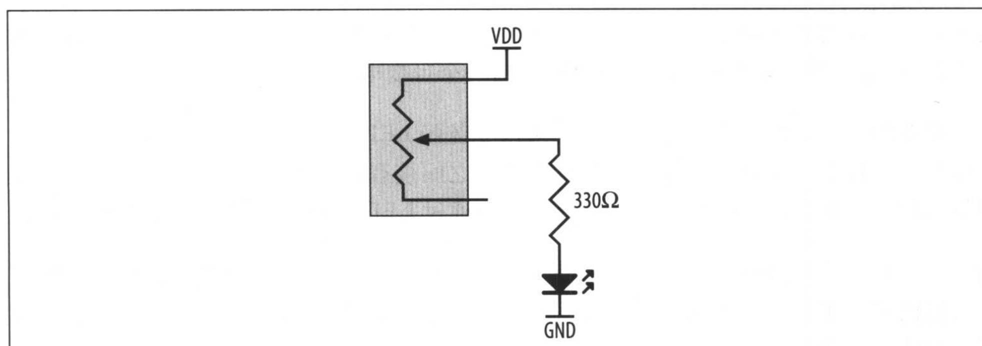


图 4-25：使用电位计来调整 LED 的亮度

电容

电阻是一个对流过它的电荷进行阻挡的元件，而电容则是用来存储电荷的器件。电容量用法拉（或更为正式的，法拉第）来度量，符号为C，单位符号为F。典型的电容从微法（μF）到皮法（pF），其值不等。

电流、电容以及电压之间的关系如下：

```
I = C * dV/dt
```

这里的dV/dt是电压相对于时间的变化率。

电路原理图中电容的表示符号如图4-26所示。左边一个是双极的，而其他两个都是单极的。单极电容有正极引线和负极引线，在电路中必须以正确的方向来放置，否则它会被击穿（单极电容上有标志来表明正负极）。双极电容不存在极性问题。

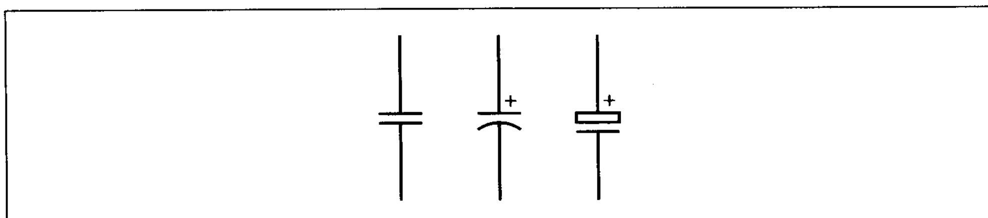


图4-26：电容符号

在电容上加上电压，它就会充电。如果撤销电压源，并且电路中某处存在电流通路，那么电容就会放电，从而提供（短暂的）电压和电流源，如图4-27所示。

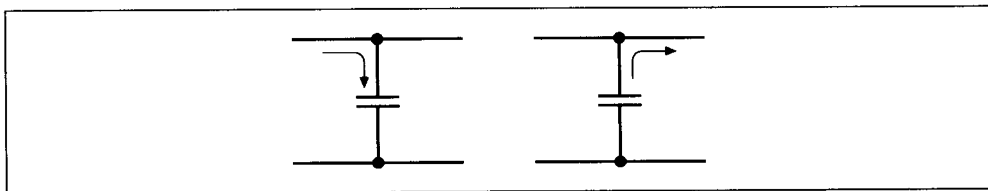


图4-27：电容充电和放电

这是极其有用的一个特性。给定的电压源可能有DC分量（固定电压）和AC分量（迭加的波形电压）。（这里的分量并不意味着是一个物理设备，而是指一个电压成分。）在电压源的DC分量作用下，电容器充电到一定程度，接着由于AC分量的作用而交替地进行充电和放电。实际上，电容最终使得AC分量的高压和低压达到平衡，其结果是把

电压源的波形电压过滤掉了。这就是所谓的电容对电压源的 AC 和 DC 分量的退耦。例如，通常使用这种方法来去除来自电源的电气噪声。

电容这一特性的相反的应用是，它可用来阻挡电压的DC分量，只允许AC分量通过，如图 4-28 所示。

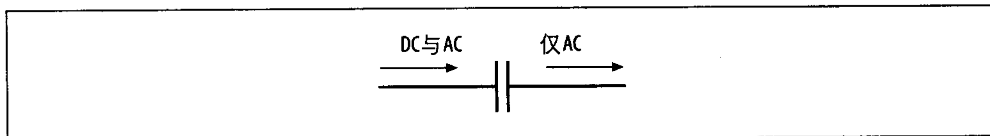


图 4-28：隔直流电容

电容也可以进行串联和并联，如图 4-29 所示。

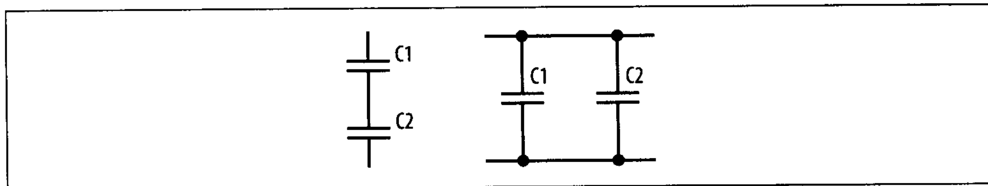


图 4-29：电容串联和并联

电容的串、并联效果与电阻的串、并联恰好相反。在电容串联时，总的电容值可由以下公式计算：

$$C_{TOTAL} = C1 * C2 / (C1 + C2)$$

并联时，总的电容值的计算公式为：

$$C_{TOTAL} = C1 + C2$$

电容的类型

电容有几十种不同的类型，每一种类型都基于不同的技术。常见的电容有陶瓷电容、电解电容和钽电容。

陶瓷电容的尺寸和容值比较小，其值从几个 pF 到大约几个 μF 不等。这种电容通常用作集成电路电源引脚的退耦电容，以及晶体电路（其他应用中的）里的旁路电容。

电解电容外表看起来像小圆柱，主要用于电源的退耦，其容值从100nF到几个F不等（这里主要讨论大电容）。不过，这种电容的精确性很差。它们的实际电容值与标称值可能

相差甚远。因此，当对电容的电容量偏差要求严格时，这种电容是不能使用的。只有在对所需的容值精度要求很低时，才使用它们。

电解电容的另外一个问题是容易老化，并且老化程度越大性能就变得越差。所以，期望在电路里始终使用这种电阻是不可能的。虽然如此，这类电容仍在电子设备中大量使用，原因之一就是这类电容很便宜。当这些电容失效时，产品也就过了保修期。然而，这类电容比一般计算机产品的平均寿命长得多。在这些电容失效之前，你早已将计算机升级到新的型号了。

注意： 旧式收音机或者高保真度收音机出现故障的原因通常是这些设备里的电解电容失效了。你经常可以在旧货市场买到非常便宜的电子设备。拿上烙铁，十分钟就可以换掉电解电容，无法工作的设备很快便得到重生，就像新买的一样。在大多数情况下，这种解决办法都是可行的。

钽电容在体积上比陶瓷电容稍微大一些，但没有电解电容大。表面贴片钽电容看起来与其他表面贴片电容非常相似：小的普通矩形。这种电容的容值从 100nF 直至几百 μF 都有。这类电容通常也是用于电源退耦，但其精度要比电解电容高得多，这就意味着它们的实际电容值与其标称值非常接近。只要可能，我的公司在产品设计时更倾向于选择使用钽电容而非电解电容。毕竟，我们希望产品能够用得更长久一些。

图 4-30 所示为一些常用的电容。

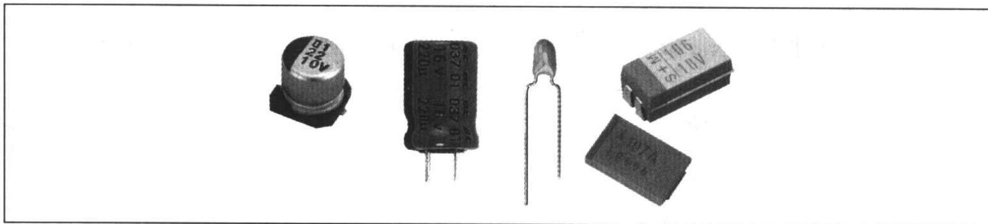


图 4-30：电容（未按比例）

RC 电路

把电容和电阻联合起来会产生一些有趣和有用的效果。一个电阻 – 电容的联合就是所谓的一个 RC 电路，它们的联合有三种形式。在第一种方式中，电阻和电容并联在一起，如图 4-31 所示。

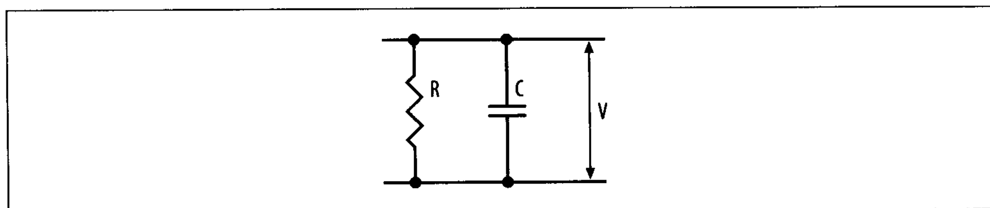


图4-31：电阻和电容并联

这种联合能用来做什么呢？施加在这两个元件上的电压（V）将对电容充电（也会有些电流穿过电阻）。当所施加的电压撤去时，电容便会穿过电阻释放电荷。电阻将限制电荷释放的速率，因为电阻限制了电流。从欧姆定律中，我们可以得到：

I = -V / R

（之所以是负电压是因为我们在让电容放电）现在，流出电容的电流可以由下面的公式得到：

I = C * dV/dt

所以，我们会得到：

dV/dt = -V / RC

对上式从时间上进行积分，结合初始条件，我们就会得到：

$V = e^{-t/RC}$

这就是电容放电时的电压波形，如图4-32所示，它描绘了电容两端电压的变化。

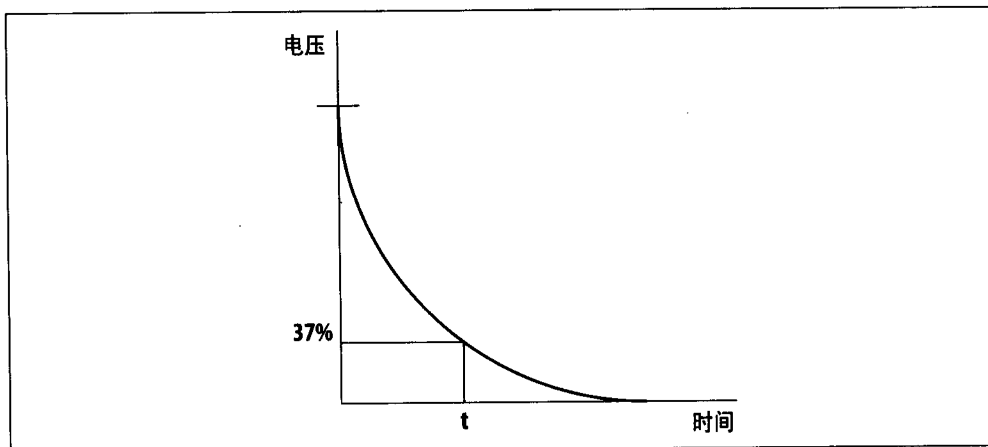


图4-32：并联RC电路放电

并联RC电路的输出电压呈指数衰减。当输出电压达到最大值的37%时，相应的时间值叫作这一电路的时间常数，是R和C的简单乘积。

```
t = R * C
```

例如，电阻为100kΩ、电容为10μF的一个并联RC电路的时间常数为1s。

第二种形式的RC电路为串联RC电路，如图4-33所示。

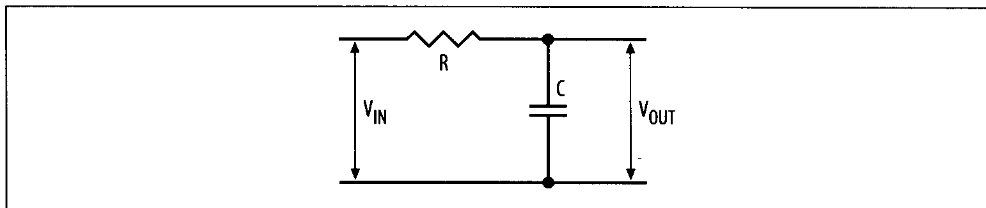


图4-33：串联RC电路

当一个电压加到RC电路的输入端时（左端），电流就会通过电阻，电容也就开始充电。然而，电阻限制了电流，进而限制了电容充电的速率。

此时，流入电容的电流由下式得到：

$$I = C * dV/dt$$

这一电流与通过电阻的电流一样大，利用欧姆定律，我们也可以得到这一电流：

$$I = (V_{IN} - V_{OUT}) / R$$

这里的 $V_{IN} - V_{OUT}$ 是电阻两端的电压差。将上述两个方程结合起来，就可得到微分方程：

$$dV/dt = (V_{IN} - V_{OUT}) / RC$$

对其积分，得到电容上的电压：

$$V_{OUT} = V_{IN} (1 - e^{-t/RC})$$

这也是一个指数方程，然而这次它意味着电容以指数方式充电。电容的电压变化波形如图4-34所示。

在这种情况下，时间常数就是电容的电压达到输入电压的63%的时刻。与并联RC一样，串联RC的时间常数也是R和C的简单乘积。

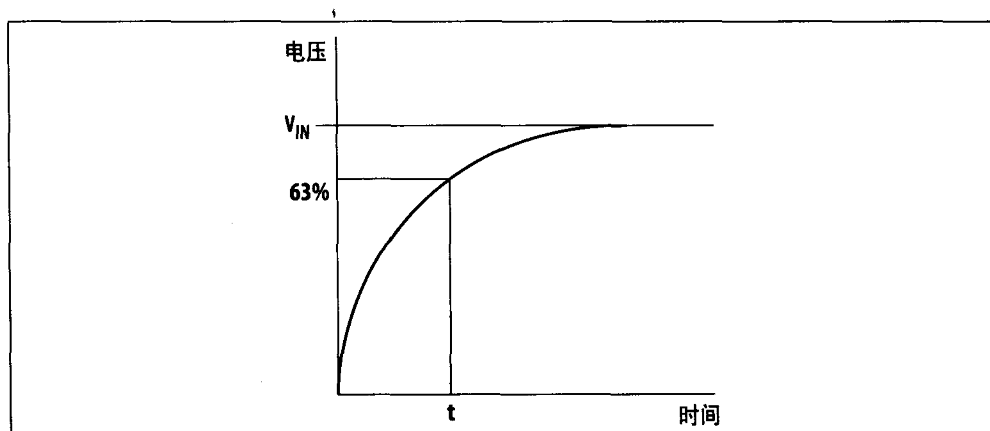


图 4-34：串行 RC 电路充电

这种类型的RC电路是一个简单的低通滤波器。这种电路能将信号的高频分量滤掉，从而在主信号里削弱它们，而低频分量没有任何衰减。这种类型的电路对于去掉叠加在信号上的高频噪声来说非常有用。

任何处理器和外部设备芯片的每个输入引脚上都会有少量的输入电容。输入电容连同电路连接固有的微小阻抗以及引脚的输入阻抗，这就意味着施加在引脚上的数字电压实际上呈指数增长，而不是（数字的）陡然上升的，如图4-35所示。这种影响很小，但在高速电路中，或者当几个设备连接到同一信号线上而所有的输入电容不可忽略时，这种影响就很大了。

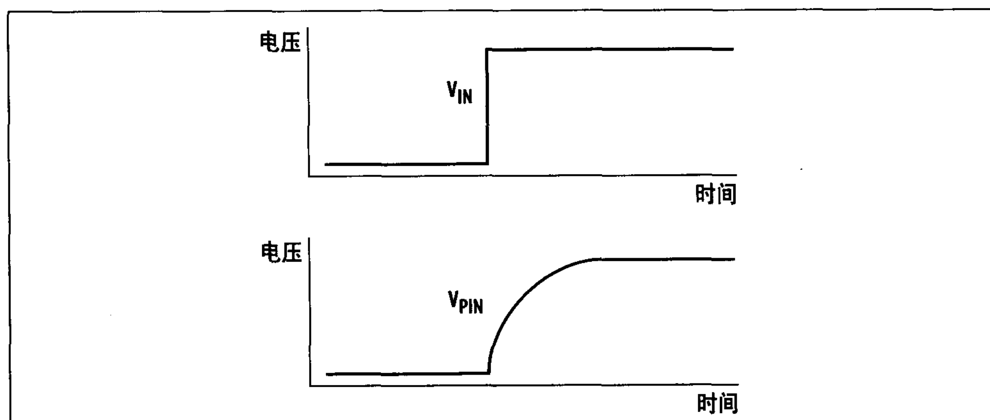


图 4-35：在芯片输入引脚上充电

输入端的感应作用会引起另外一个特性，如图4-36所示。当源电压突然变化时，电感应作用导致“阻尼振荡”。我们很快就会讨论电感。

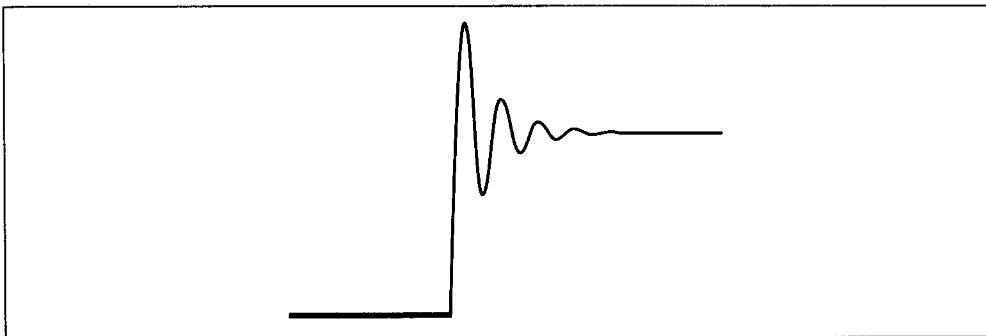


图4-36：电感应作用引起信号输入的阻尼振荡

第三种形式的RC电路如图4-37所示。

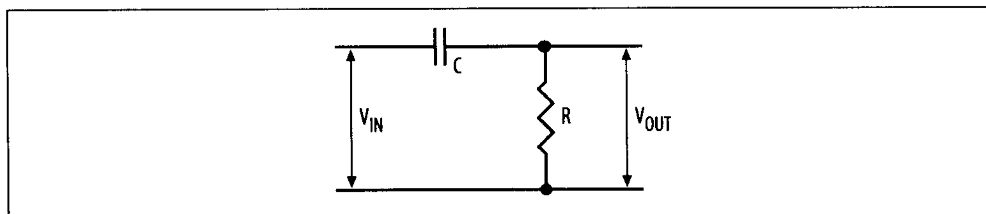


图4-37：RC滤波器

这种类型的电路是一个简单形式的高通滤波器，因为它只输出高频信号。这种电路里的电容通常被称作隔直流电容。

电感

电感是无源器件，实质上就是一个导线卷。电感的电路图符号如图4-38所示。感应系数的单位是亨利，符号为L，单位符号是H。

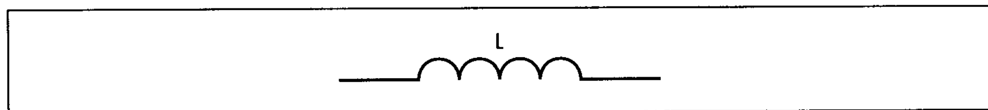


图4-38：电感电路图符号

电感上的电压通过下列关系式改变着通过电感的电流：

```
V = L * dI/dt
```

向电容中加入电流会引起电压产生，相反，向电感中加入电压也会产生电流，因之而产生的能量以磁场形式存储在电感里。当撤销电压时，磁场便消失，所存储的能量转变为峰值电压。

图 4-39 所示为 R-L 的串联电路。

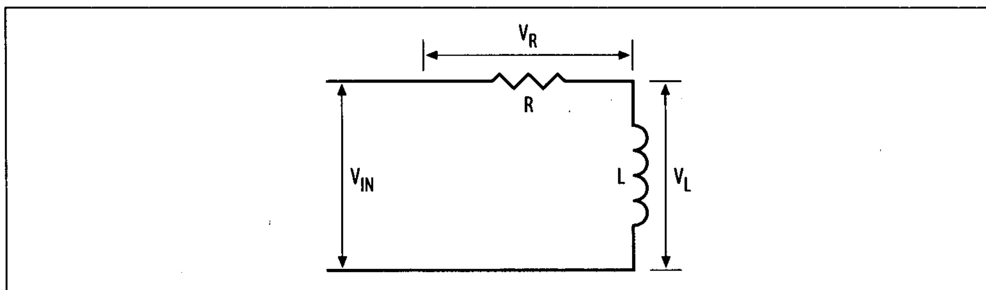


图 4-39：串联 R-L 电路

电阻两端的电压（V_R）和电感两端的电压（V_L）如图 4-40 所示。当在 V_{IN} 加上电压时，电阻的初始电压很小，而电感的电压则很大。随着电流穿过电感，电阻两端的电压逐渐增加，而电感上的电压相应地减少。

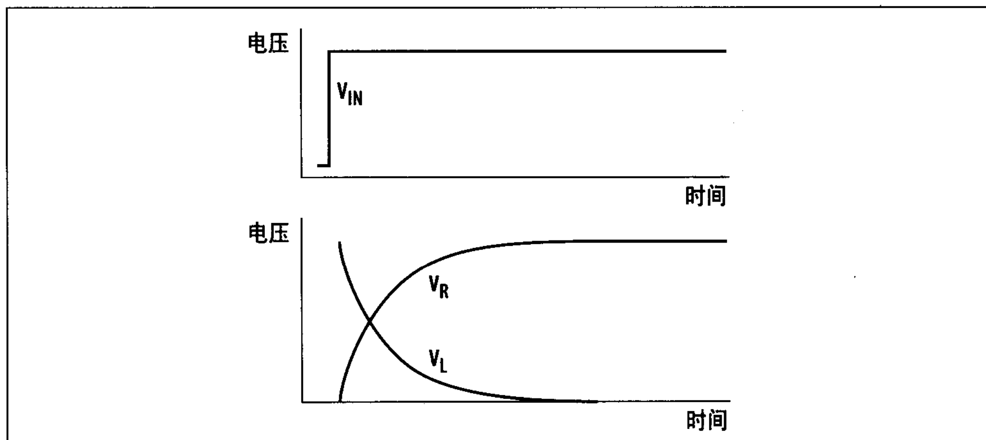


图 4-40：串联 R-L 对阶式信号输入的响应

图 4-41 所示为 R-L-C 串联电路。

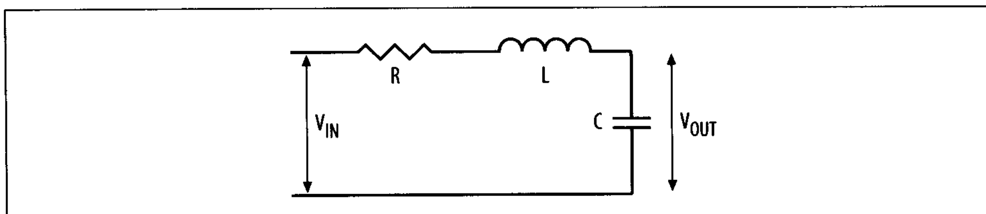


图 4-41：串联 R-L-C 电路

R-L-C 电路对阶式信号输入的响应如图 4-42 所示。

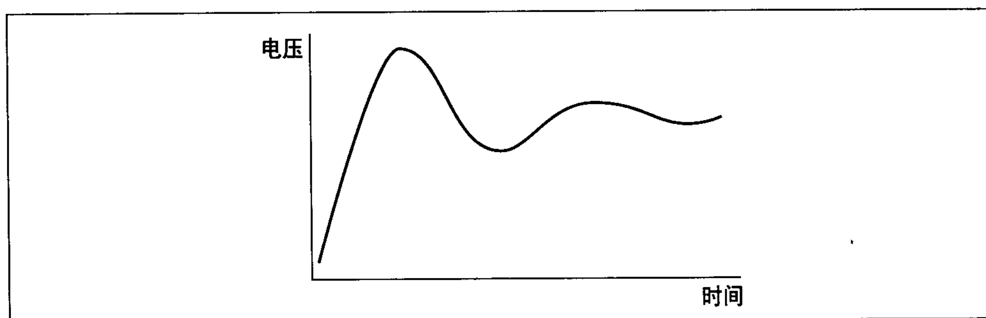


图 4-42：R-L-C 电路的响应

图 4-43 所示为所有器件并联的 R-L-C 电路。

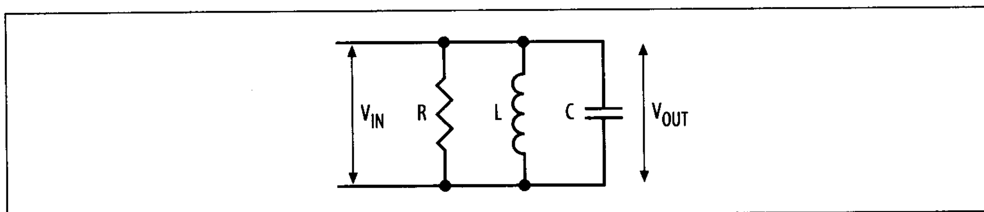


图 4-43：并联 R-L-C 电路

并联 R-L-C 电路对阶式信号输入的响应如图 4-44 所示。

电感一般用在切换式稳压器（见第 5 章）中，也可用作滤波器（与电阻和电容结合），以去除信号中不想要的频率分量。感应作用存在于许多器件中，并且感应电压峰值是嵌入式系统设计者的梦魇。

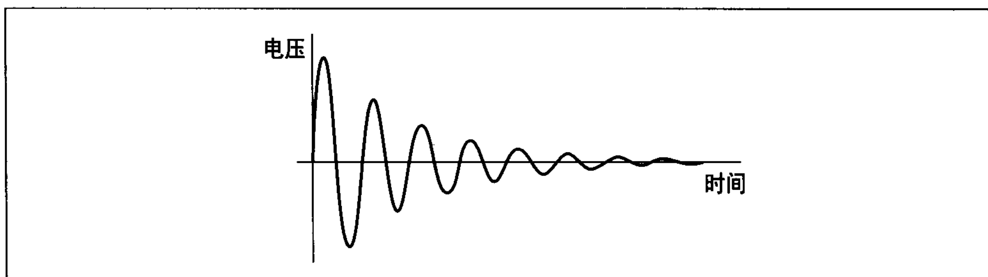


图 4-44：并联 R-L-C 电路响应

变压器

变压器与电感有关。一个变压器有两个紧密耦合的金属线圈，分别叫作主线圈和从线圈。变压器的电路图符号如图 4-45 所示。

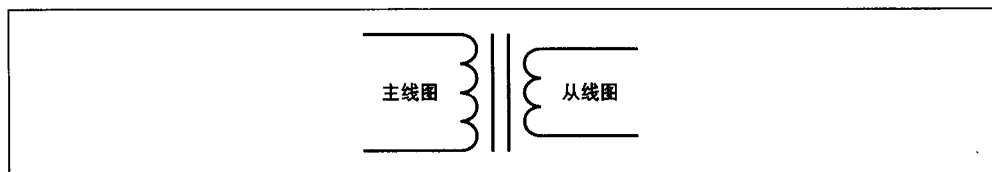


图 4-45：变压器电路图符号

通过主线圈的 AC 电流会产生一个电磁场，该电磁场的磁力与主线圈的圈数成比例。因为从线圈处于磁力范围之内，磁场就会产生电流穿过从线圈（所以也会产生电压）。由于从线圈与主线圈的线圈数不同，主线圈所产生的磁场就会在从线圈里引起大小不同的电压和电流（从线圈是电路的一部分，当然会有电流产生）。因此，变压器可以看作是电压倍增器（或分配器）。主从线圈的线圈数的比率决定了电压的放大倍数。

由于变压器通常特别有效，因此主线圈里的大部分功率都可以转到从线圈中。如果从线圈增加了主线圈的电压，那么从线圈的电流就会相应地比主线圈小。相反地，如果从线圈的电压比主线圈的小，那么从线圈的电流就会比主线圈的大。

变压器通常用在电源里面，把高电压（110VAC 或 240VAC，具体取决于你住在哪）转变为电子系统和其他设备所用的低电压。它们也用来隔离高压供电系统和动力电系统。

比率为 n 的变压器，其阻抗增值为 n^2。因此，变压器的另外一个用途是提供一种改变传输线阻抗的途径。例如，一个以太网端口会在接口芯片和电缆之间使用一个变压器。

二极管

二极管是极其有用的半导体设备。它们有很有趣的特征，即能在一个方向上通过电流而在另外一个方向上截止电流。二极管可用来允许电流从一个部件流到另一个部件，而禁止电流“倒流”到你不希望的地方。

电路原理图中的二极管符号如图4-46所示，图中的箭头表明了传导方向。箭头方向表明二极管的阳极，或者说正极；而条栅表明二极管的阴极，或者说负极。图4-46中二极管左侧的高电压将允许电流到达右侧；然而，图中二极管右侧的高电压将阻止电流流向左侧。

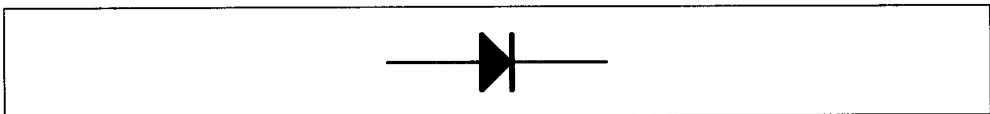


图4-46：二极管的电路图符号

导通时，二极管有一个正向电压降，这意味着在二极管的阳极和阴极之间存在一个电压差。例如，一个二极管可能有一个0.7V的正向电压降，如果这个二极管是某个大型电路的一部分，并且阳极的电压是5V，那么阴极的电压就是4.3V。

二极管具有多种形状和大小。图4-47所示为一个小型的功率二极管。图中标注的白线部分代表二极管的阴极（负极）。

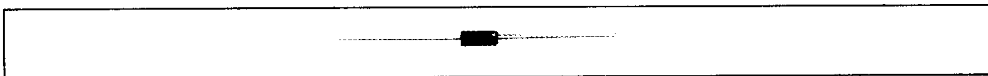


图4-47：功率二极管（1N4004）

二极管对于去除信号源里的负电压非常有用，这个过程叫作整流（rectification）。四个二极管可以组合在一起形成一个桥式整流器，如图4-48所示。这个桥式整流器“翻转”了波形的负分量，因此在输出端只有正电压。在输出端上的电容器可以用来平滑整流后的波形。

这种结构通常作为嵌入式计算机和其他数字系统的电源输入。一个电压可以施加在左端的输入端，而不必考虑哪个是正极哪个是负极。桥式整流器可以确保右上端输出的是正电压，与此同时，电流流过左端任一极接负电压的桥式整流器，然后从右下端返回。

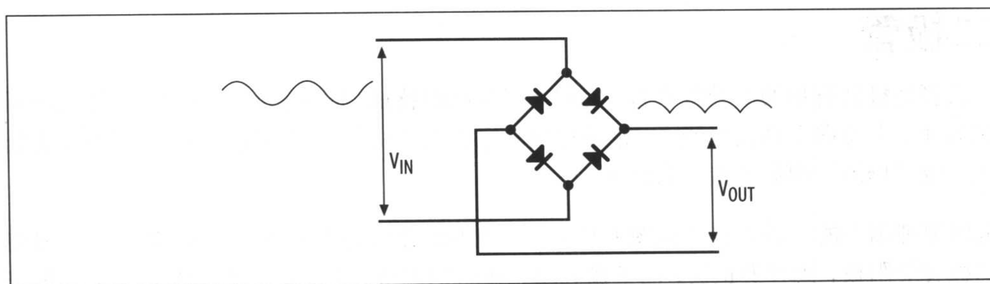


图 4-48：桥式整流器

最为常见的二极管就是发光二极管（Light-Emitting Diode，LED），电路图符号如图 4-49 所示。图 4-50 所示为直插式 LED。所有二极管工作时，都会产生少量的光（虽然由于二极管的外表封装你通常看不到），而 LED 非常善于利用这种光。

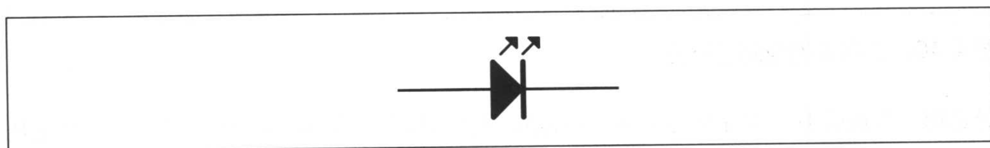


图 4-49：LED 的电路图符号

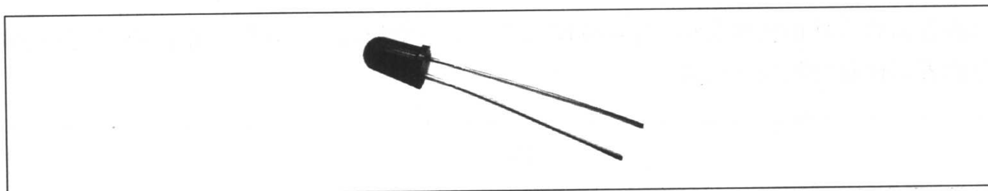


图 4-50：LED（5mm 直插式）

通过 LED 的电流量有个限制，超越这一限制就可能损害或者破坏 LED。因此，使用时 LED 一般与限流电阻相连，如图 4-51 所示。有些 LED 内部集成了一个限流电阻。可是大多数LED没有这样做，因此在使用LED时，查看生产厂商的数据表是非常重要的。通常需要给 LED 接上一个电阻，其电阻值很容易算出。

假设LED的正向电压降为1.6V，额定电流为36mA。我们需要选择一个电阻来限制通过LED的电流为这一额定值。在我们的电路里，LED和电阻是串联的，它们两端的总电压是 5V。因此，如果 LED 的电压降为 1.6V，那么很容易就可以计算出电阻上的电压：

```
VR = 5 - 1.6
   = 3.4V
```

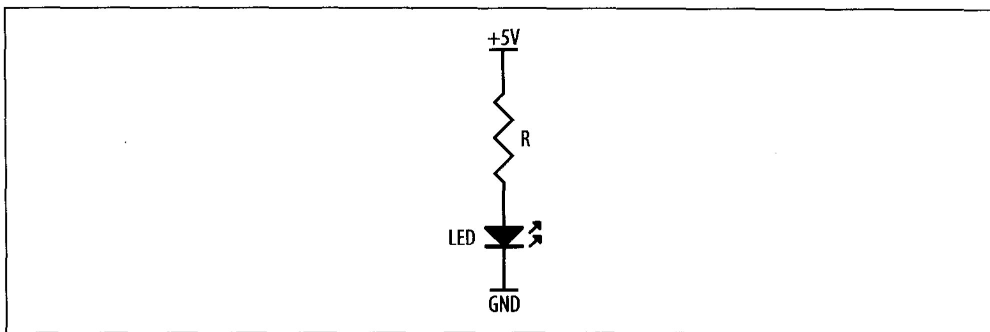


图4-51：带限流电阻的LED

这样，如果电阻上的电压为3.4V，我们需要限定的电流为36mA，通过欧姆定律就可以计算出电阻值：

```
R = V / I
  = 3.4 / 0.036
  = 94.44Ω
```

一个100Ω的电阻就可以很好地实现上述要求，LED便会发出明亮的光。如果你想降低LED光的强度，只需使用阻值大一点的电阻来限定通过它的电流。需要注意的是，36mA是LED能够接受的最大电流值，因此总是需要使电流低于36mA的电阻。所以，我们总是选择比R（=94.44Ω）值大的电阻。

电阻所消耗的功率由下式可得出：

```
P = V * I
  = 3.4 * 0.036
  = 0.1224Ω
```

可选的电阻有着不同的额定功耗。选择一个恰当额定功耗的电阻很重要。在本例中，我们使用一个0.125Ω的电阻。

在家用电器中所见到的电源指示LED就是以这种方式工作的。这个简单的LED电路（或其变种）驱动着PC机面板、VCR和DVD播放机、蜂窝电话以及其他电器上的LED。LED阵列正在代替传统的灯泡用作交通灯和铁路信号灯，因为LED寿命长且亮度高（单位面积上）。

可选的LED有红、绿、黄、蓝和白几种颜色，蓝色和白色的LED生产起来比较困难，因此价格也比较贵。

了解另外两种类型的二极管可能比较有用，它们是齐纳二极管（Zener）和肖特基二极管（Schottky），如图4-52所示。

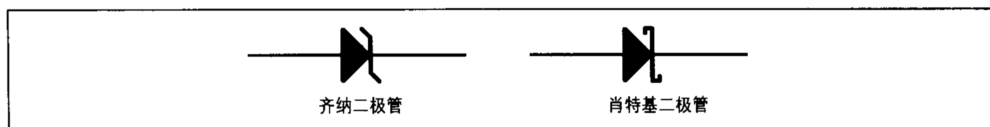


图 4-52：齐纳二极管和肖特基二极管

齐纳二极管呈现一种叫作动态阻抗（dynamic resistance）或者小信号阻抗（small-signal resistance）的特性。当通过齐纳二极管的电流改变时，电压却并不改变。实际上，齐纳二极管可以充当可变电阻，其阻值依赖于电流。齐纳二极管通常用于提供参考电压，如图 4-53 所示。

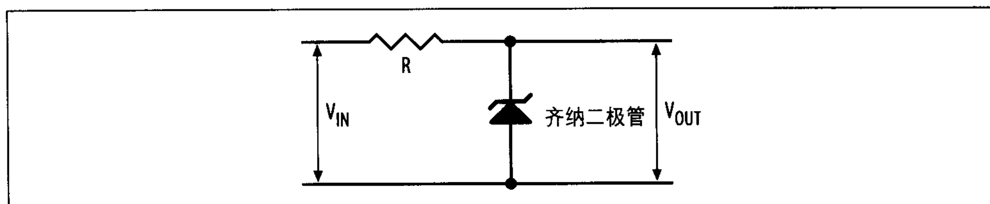


图 4-53：通过齐纳二极管来提供参考电压

由欧姆定律，我们得到：

$$(V_{IN} - V_{OUT}) = I * R$$

现在，如果改变 V_{IN}，从逻辑上讲电流也会相应地改变。因此我们可以修改上述方程为：

$$(\Delta V_{IN} - \Delta V_{OUT}) = \Delta I * R$$

齐纳二极管充当了动态电阻源（dynamic resistance），用符号 R_d 来表示。现在我们就有了一个有效的电压分配器。因此计算 V_{OUT} 的方程为：

$$\Delta V_{OUT} = \Delta V_{IN} * R_d / (R + R_d)$$

肖特基二极管也叫载流二极管（hot-carrier diode），表现与传统二极管相似，但其正向电压降极小。出于这个原因，这类二极管通常用于电源电路和信号整流。

晶体

在元器件之旅的最后一站，我们来看一下晶体。正如其名，晶体是小石英块（即二氧化硅）。石英晶体是一种称为压电的材料。这种物质在受力（压缩、拉伸和扭曲）时会产生一个电压。这种效果被应用在了麦克风里。声音使压电材料产生振动，这种材料便会

产生一个小的交流电压，电压的大小直接正比于声源；接着就对这一电压进行放大以便传播、记录或处理。

对压电材料采取相反的作用就会产生相反的效果。在压电材料上施加电压，它就会扭曲或者振动。有些扬声器就是使用这些压电材料来发出声音的（然而，大多数采用了其他一些技术，如电磁学）。大多数蜂鸣器能够发声都是基于压电材料的特性。

现在，有关石英的一个巧妙的事实是：一定大小的石英块会以一定的（且固定的）频率振动。由于这个原因，石英可以用作振荡器以产生一个正弦波，反过来又能为微处理器和其他数字电路产生时钟信号。几乎每台计算机系统在其主板的某个地方都有一个或两个晶体，用于产生最终驱动整台机器的定时。这个晶体只是一块石英而已，它的一端镀了金属，与金属线相连，封装在金属罐里。

晶体的电路图符号如图 4-54 所示，晶体的外观如图 4-55 所示。

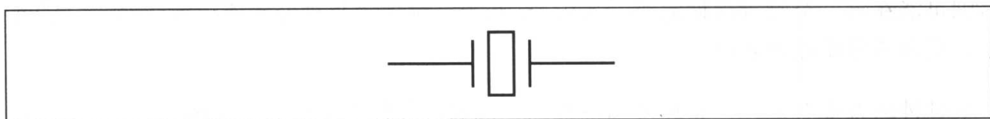


图 4-54：晶体的电路图符号

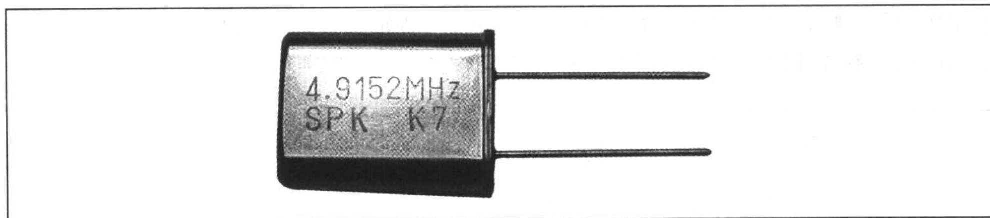


图 4-55：晶体

晶体工作时需要一个驱动电路。这些晶体有些变化无常，由于种种很难跟踪的影响，它们并非总是以你所期望的频率振荡。幸好，对于这个问题有两个解决办法。方法之一是大多数处理器（以及其他需要定时的芯片）有内部振荡电路。你所需要做的只是添加一个外部晶体（还有一个或两个电容器），系统即可很好地工作起来。对于其他芯片，操作起来并非如此简单，你需要一个包含晶体和驱动电路的完整的振荡模块 —— 这就是方法之二。这种情况下，你要做的是向它们提供电源和接地，它们也会很好地运转起来。

时钟和振荡器

所有的微处理器（以及其他少数数字设备）都需要时钟。所谓的时钟，就是来自振荡器

的输出，这一输出管理着处理器以及其他与时钟有关的所有系统事件（到此，你会疑惑，这个时钟与每天的时间没有任何关系呀。其实，你把时钟看作是一个数字脉冲流即可）。时钟的频率通常用kHz、MHz（1000kHz）或GHz（1000MHz）来表示。一个微处理器的时钟频率也叫作它的时钟速率。

一个给定的处理器将有一个最大和最小时钟频率。这表明了振荡器驱动处理器的频率范围。我们说最小时钟为零的处理器就具有静态操作和DC操作。这意味着处理器能够使时钟停下来，也能在稍后的时刻重新启动操作而不受什么影响。如果处理器的最小操作频率大于零值，那么称该处理器具有动态操作。如果振荡器的频率低于上述动态处理器的最小频率值，那么处理器的寄存器的内容有可能会被破坏掉。

处理器的时钟速率与处理器以怎样的速度执行软件有关。以快速时钟运行的处理器就比以慢速时钟运行的同类处理器执行软件的速度快。但时钟速率并非衡量处理器速度的唯一标准。一种处理器体系结构可能采用32个时钟周期来执行一条指令，而另外一种处理器则可能在每一个时钟周期完成一条指令。因此，尽管这两种处理器运行于同一时钟速率，后者明显要比前者快。

产生时钟的方式有多种，而具体哪种方式合适则很大程度上取决于你所使用的处理器。有些处理器期望一个数字（方波）时钟输入。对于以通用频率运行的处理器，大多数处理器都属于这类，最好的选择是采用一个名为振荡器模块的设备（如图4-56所示）。该设备有四个引脚，以给定的频率提供了一个方波时钟输出，而所需的只是电源和接地。这就简化了系统设计，因为它们是即插即用的设备。

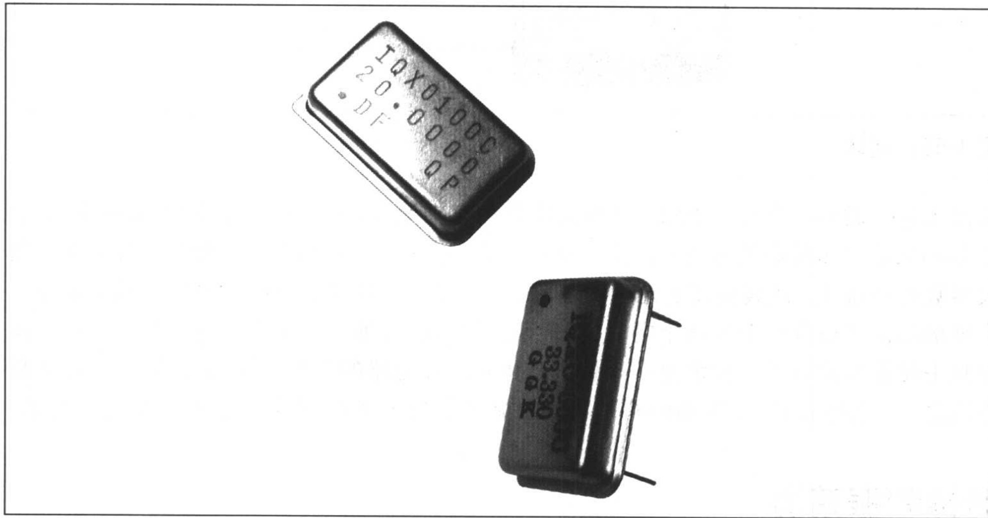


图 4-56：微处理器振荡器模块

许多处理器（包括所有我能想到的微控制器）都包含振荡器电路，并且通常只需要增加一个晶体和旁路电容即可，如图 4-57 所示。这里的电容用于去除来自振荡的高次谐波。

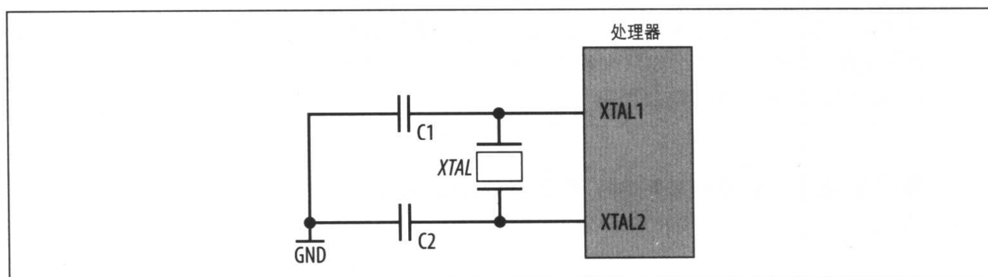


图 4-57：内部振荡器的晶体电路

功率与速度

设计一个系统时通常必须使其功耗最小。低功耗可以降低系统产生的热量，或使系统具有便携性。通过设备的电流越大，它所产生的热量就会越多。过多的热量能导致系统停止运转。虽然可以在系统中添加冷却子系统和温度监控器，但更好的办法还是简单地降低系统的功耗继而降低系统所产生的热量。对于以电池供电的系统，所需电流越低，电池的寿命越长。由于上述原因，低功耗设计是极为有利的。

数字系统的大部分电流的使用发生在状态转变期间，换句话说，也就是发生在时钟边沿期间。时钟沿越频繁，系统的平均电流使用也就越高。因此，处理器越快，它的功耗越大。相反，处理器的运行频率越低，功耗也就越小。很多嵌入式处理器都是低功耗的。不同体系结构的处理器的功耗也可能不同。一个以与Pentium处理器同样时钟频率和工作电压运行的 ARM 处理器功耗却相当低；正是由于这个原因，ARM 处理器通常用于 PDA 设备中。

使用带静态运转的设备会允许系统的时钟减慢（或停止），且由于系统功耗直接与运转速度相关，因此这也降低了机器的总体功耗。时钟可以以几种不同的方式减慢。第一种是时钟永久性保持慢速。对于不需要大量计算处理能力的应用（如红绿灯控制器），时钟频率为 32kHz（而不是 20MHz）的处理器可能就够用了。

对于偶尔需要繁重计算的系统，在处理器空闲时系统时钟可以减缓。许多便携式计算机采用这种技巧来降低其功耗。当使用它时，处理器全速运转，如果系统待机超过 30 秒，系统时钟就降 kHz 范围内。如果系统继续待机几分钟，时钟就降为 Hz 范围内。由于这期间用户并不用机器（这也就是所谓的待机），所以感觉不出任何差异。一旦需要处理器来完成一个任务（如有键按下），时钟就切换回全速，机器又回到正常运转状态。

注意： 掌上电脑非常有效地使用了这种技术。这种机器是事件驱动的，也就是说当用户触摸屏幕时或者当一个I/O设备需要服务时，机器就做一些事情。因此，在事件之间，处理器的速度大大降低。当一个事件产生时，系统就切换回全速状态对事件进行处理，事毕又一次回到待机状态。由于处理器的待机状态比工作模式的时间长得多，所以系统可以最小限度地使用电池。这也是为什么你从来没有看到（也不可能看到）在掌上电脑上进行计算密集的应用。否则，电池很快就会用完了。

有些计算机系统甚至完全停止时钟，直到有处理器请求产生，此时，通过外部设备激活系统时钟。

许多处理器具有SLEEP和HALT模式，这可以降低处理器功耗。有些处理器把上述模式扩展为SLEEP、NAP、DOZE、SNOOZE等模式，每一模式具有不同级别的功耗使用，每一模式都需要不同的时间来“唤醒”.处理器（处理器睡眠越沉，即处理器所处的模式功耗越小，唤醒处理器操作的时间就越长）。

到此就结束了对电子元件的讨论。我没谈到的一个主要类型的元器件就是晶体管。之所以未在此提及，是因为在嵌入式系统里这种器件一般比较少见，况且即便是适当提及晶体管，也必会占用很大的篇幅。如果你有兴趣学习晶体管，有许多优秀图书可供参考，只需到当地技术类书店看看或浏览因特网即可找到这类图书。

数字信号

作为一个电子电路，计算机的操作是与电压和电流有关的。如果你打算制造出可以运行的系统，那么对计算机内的电压和电流的基本原理的理解是必不可少的。计算机内常见的工作电压是5V或3.3V。对于一些低功耗或特别快的计算机，工作电压可能小到1.8V甚至更低。

一个数字设备的输出引脚可能处于下列三种状态之一：高电平（逻辑1）、低电平（逻辑0）或三态（高阻态，也叫漂移）。当引脚上的输出电压高于一个给定阈值时就定义为逻辑高电平。当一个设备的引脚输出为高时，这一引脚就对其连接提供电流。同样地，当输出电压低于给定阈值时，我们称设备的引脚吸收电流。典型情况是，元件吸收的电流多于它所提供的电流。

一个三态引脚的输出既不是高电平也不是低电平，而是高阻态，这样流入或流出该引脚的电流都被忽略。实际上，流入或流出该引脚的电流对于与这一引脚相连的其他元件而言是不可见的。例如，计算机系统内可能有好几个存储设备连接到数据总线，当从一个设备读取数据时，它的数据输出要么为高电平要么为低电平（对应于读回的比特模式）。

系统内所有其他的存储设备，由于它们未被访问，因此会把它们的数据总线置为三态，不会参与处理器和被访问存储器之间的读数据处理。

高电平的阈值和低电平的阈值随着设备类型的不同会有所不同。为了让输入设备确认所给信号是低电平还是高电平，输出设备必须提供在适当限度内的信号。阈值可以不同，但同一逻辑系列的阈值必须一致。回顾过去，逻辑系列的数量有限，一个系列内的每一设备都遵循该系列的阈值，所以设计数字系统的工作是很容易的。而如今，既希望设备的功耗低，又希望设备用途尽可能地多，这就使得设备高电平和低电平的阈值多种多样。因此，一个特定芯片的输入端的低阈值就可能与它所连接芯片的输出端的低阈值不匹配。所以，你必须查看所使用的所有元件的数据手册，确保它们可以一起工作。

电压阈值

TTL（Transistor-Transistor Logic，晶体管－晶体管逻辑）系列的电压定义：低电平，输入最高电压为0.8V，输出最高电压为0.4V；高电平，输入最低电压为2.0V，输出最低电压为2.4V。许多处理器认可TTL输入，虽然实际上只有极少数处理器是TTL兼容设备。这点很重要，你不能假定一个输出或输入会在电压规范之内。例如，一个老的80386处理器时钟输入的最低高电压在20MHz时是4.2V，在25MHz时是3.7V，这些电压就明显比TTL标准高。如果用一个标准TTL设备来驱动该处理器的输入引脚，TTL设备可能并不能产生足够大的电压让处理器认为是高电平。这一例子的教训就是查看产品手册！这正是器件的电气（和时序）规格列在产品手册里的原因。

有许多不同的逻辑系列，每一系列都有其自身的阈值电压和其他特征。除此之外，有一些元件不属于任何特定的逻辑系列。如果是这样，产品手册就是指导你如何处理的最佳指南。

当设备输出一个高电平，且其输出电压大于输入设备的高态阈值时，从输出引脚到输入引脚就会有电流通过。这个输出设备就提供电流，而这个输入设备就吸收电流。

与此相反，对于某些类型的数字逻辑，当一个设备输出一个逻辑低电平，且这一输出电压低于输入设备的低电平阈值时，虽然输出设备控制着电压，但电流还是从输入引脚流入输出引脚，此时，输出设备吸收电流，而输入设备提供电流，如图4-58所示。

电流的量非常重要。一个给定设备会对其所能提供或吸收电流的多少有所限制。超越这一电流限制就可能永久性地损坏集成电路。因此，计算你的系统里的电流以确保它满足所有的需求是很重要的。

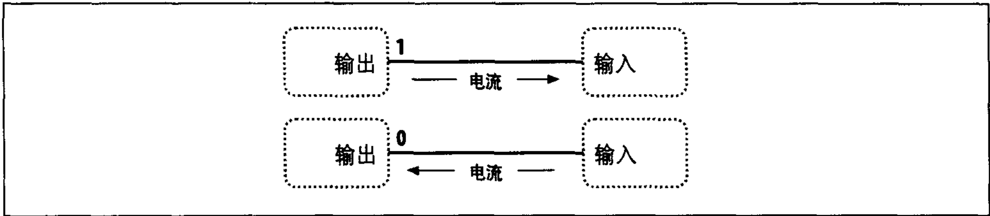


图 4-58：数字设备间的电流

电气特性

在任何芯片的技术手册中都可以看到三部分的内容：最大绝对额定值（Abolute Maximum Ratings）、直流电气特性（DC Electrical Characteristics）以及交流电气特性（AC Electrical Characteristics）。这些数据相当重要，但通常鲜为人知。所以让我们在本节中了解一下它们的含义，以及如何利用这些信息来制作一个可靠而又高效的嵌入式系统。首先介绍最大绝对额定值。

最大绝对额定值

首先你将会在“电气特性”部分中看到标有“最大绝对额定值”的部分。芯片被设计成可以在某些特定条件下操作，在正常操作期间建议不要逾越这些条件。如果真的逾越了这些条件，会产生怎样的后果呢？这就好比说，你在何种状况下真的会将芯片毁坏？最大绝对额定值就是在告诉你这个问题。最大绝对额定值指出，在芯片被毁坏之前可以给它多大（有时是多久）的负荷。

表 4-1 所列为达拉斯半导体（Dallas Semiconductor）的 DS87C550 微控制器的最大绝对额定值。

表 4-1：DS87C550 微控制器的最大绝对额定值

参数	额定值
相对地线任何引脚电压值	−0.3V~（VCC +0.5V）
相对地线 VCC 电压值	−0.3V~ 6.0V
操作温度	−40℃ ~ +85℃
存储温度	−55℃ ~ 125℃
焊接温度	160℃，10 秒内

上表中每个参数的含义是什么呢？第一个参数的含义为：此控制器任意引脚所能承受的电压范围。一般来说，加在引脚上的电压（作为信号或电源），其范围从零（接地）到VCC（以DS87C550为例，就是+5V）。这个最大绝对额定值指出，如果引脚上的电压低于−0.3V（换言之，就是一个负电压），此芯片就可能被毁掉。同理，如果引脚上的电压超过VCC+0.5V，那么该芯片也可能毁坏。现在，有许多人把VCC+0.5V解读成+5.5V，因为VCC通常是+5.0V。但有时未必如此。VCC+0.5V意味着比实际VCC（芯片电源）电压高0.5V，而不是说比5.0V高0.5V。如果供给DS87C550的电源是+4.5V（决对没有问题），那么每个引脚的最大绝对额定值就不是+5.5V而是+5V。负责设计的工程师（也就是你）必须确保连接此微控制器的装置不会超过这些参数。

注意： 注意，并非所有芯片的输入电压的最大绝对额定值都像DS87C550那样严格。我就知道一款存储芯片，尽管它是在+2.7V的电源下进行操作的，不过它却能承受高达+6.0V的输入电压。同样地，别忘了为你的每个设备查阅其技术手册。

让我们来看一个例子：假设有两个嵌入式系统，这两个嵌入式系统都使用87C550微控制器，将这两个系统合并成一个较大的系统。第一个系统是由A公司所设计，它的87C550使用+4.5V的电源。第二个系统是由B公司所设计，它的87C550使用+5.5V的电源。这两种电源电压都是正确的，而且也完全符合其技术手册的要求。当第二个嵌入式系统的一个输出为高电平时，它的电压将（可能）会是+5.5V（技术手册只是提到，当输出为高电平时，它的电压至少为+2.4V，这意味着实际上它的电压可能介于+2.4V~VCC之间）。但是，第一个嵌入式系统的微控制器的输入为高电平时，它的电压却不可以超过VCC+0.5V：

```
Max Input = VCC + 0.5V
          = 4.5V + 0.5V
          = 5V
```

所以，第二个嵌入式系统的输出电压非常有可能会超过第一个系统的最大输入额定值，即使它们之间使用的是相同的芯片。这说明了小心检查与评估系统中每个元件的重要性。如果一个工程师只因为这两个系统使用了相同的微控制器，就假定它们可以一起工作，那就大错特错，很有可能会将第一个系统毁掉。

第二个参数的含义为：此芯片所能承受的最大电源供给。以这个处理器来说，尽管它的正常操作电压是+5V（列在技术手册的其他地方），然而它实际所能承受的最大电源供给是+6V。当电源引脚上的电压超过+6V时，这将有可能危害及芯片。同样地，为电源引脚提供的负电压也将危害到芯片。

第三个参数的含义为：此芯片可以稳定运行的温度范围。如果温度低于 −40℃，此芯片会完全失效，或是让执行中的软件终止。同样地，如果微控制器运行在温度超过 +85℃ 的环境中，那么芯片也将会失效。操作温度不同于存储温度（表中的第四个参数）。此芯片可以可靠地保存在温度介于 −55℃~ +125℃的环境中，而不会对芯片造成潜在的危害。

注意： 在某些技术手册中，你将会看到 Operating Temperature（操作温度）被表示成 Junction Temperature（结温），用来代表半导体里面的温度。两者的实际含义相同。

最后一个参数的含义为：此芯片可以承受的最大焊接温度。此例中，它可以承受+160℃的温度 10 秒的时间。目前，电烙铁可以达到的温度不止 160℃。事实上，此芯片或许可以承受更高的温度，只不过它所能承受的时间将变得更短。所以，如果我要用烙铁焊接该芯片，最后一个参数指出，我必须迅速地焊接每个引脚到位。将热的烙铁靠在每个引脚数秒钟将会提升芯片的温度，并且让功能复杂的硬件电路变成电路板上毫无用处的装饰品。

直流特性

直流特性指出了芯片在正常运行期间的电压与电流。在技术手册中，通常会看到直流特性（有时包括最大绝对额定值）被划分为商用级（Commercial Grade）、工业级（Industrial Grade）或军用级（Military Grade）。这些特性指出了特定元器件的健壮性以及抗恶劣环境的能力。一个军用级的元器件要能承受得起较高的温度，具有较佳的抗干扰能力以及较长的使用寿命。有时军用级的元器件还要承受得起放射性的考验，也就是说，被放射线照过之后，它还可以继续存活下来并且继续运行。军用级元器件的缺点是它的价格比较贵，有时还很难获得，而且它的运行速度可能不如相应的商用级元器件那么快。所以你可能觉得，除非要设计军用级的硬件，否则将不会使用军用级的元器件。然而，如果你所设计的设备将会用在太空、高海拔、北极或南极这种地方温度变化急剧，而且放射线的强度也很高的地方，军用级元器件可能是比较好的选择。

直流特性的数据会以表格的方式呈现。表 4-2 所示为（一个假想芯片的）直流特性的典型数据项。这里用到“典型”这个词，我采用的是它最广义的意思。每个芯片制造商提供数据的方式各不相同，但这并不是问题，一旦你知道自己要找的是什么之后，一切都可以迎刃而解。

表 4-2：直流特性示例

参数	符号	最小值	典型值	最大值	单位
工作电压[a]	VDD	3	-	5.5	V
输入低电压	VIL	VSS	-	0.15VDD	V
输入高电压	VIH	0.25VDD	-	VDD	V
输出低电压[b]	VOL	-	-	0.6	V
输出高电压[b]	VOH	VDD − 0.7	-	-	V
工作电流[c]	IDD	-	0.8	1.4	mA
休眠电流[d]	Isleep	-	10	25	nA
输入阻抗				TBD	pF

a. 以 **VSS** 为参考点

b. 3V ≤ **VDD** ≤ 5.5V

c. 振荡器 =44MHz

d. 振荡器 =32kHz

第一个参数（工作电压）告诉我们，使用该芯片时所需提供的电压。符号（Symbol）栏只是一个标签，用于在技术手册的其他地方引用该参数。本示例中，工作电压的最大值为5V，最小值为3V，这意味着此芯片可以运行在所指出的电压范围内。典型值栏空白，代表在此范围内没有优先工作电压。有时你会看到某个芯片的工作电压最小值为4.5V、最大值为5.5V、典型值为5V。这意味着此元器件需要5V的工作电压（典型值栏的值），但可以承受 ± 0.5V的波动。有些制造商会以Nom（Nominal）来代表这个栏，它的意思跟Typ（Typical）是一样的。附注a指出，工作电压是以**VSS**为参考点。这看起来似乎平淡无奇，但却很重要。**VSS**通常会接地（零伏），但不一定都是如此。在某些特殊情况下，可能会设计成将**VSS**接上电压，而不是接地。因此，附注指出工作电压必须比**VSS**高3V~5.5V。所以，举例来说，如果**VSS**是2V，则工作电压将会是5V~7.5V（别忘了，这是电压差，这很重要）。

接下来的两个参数（输入低电压和输入高电压）告诉我们，当寄存器的输入引脚为低电平（0）或高电平（1）时分别需要何种电压级别。注意，这些都不是绝对值，而是相对于**VSS**和**VDD**而言的。

输出低电压的最大值为0.6V，但没有最小值或典型值。这意味着，当此芯片输出低电平时，它的电压将不会大于0.6V。尽管没有提供最小值，不过你可以假定最小值为**VSS**。此芯片的输出电压将不会低于此值。此参数与下一个参数的附注指出，当此芯片的工作电压介于3V~5.5V之间时，所提供的值才有效。此电压范围通常被写成3V≤**VDD**≤5.5V，代表**VDD**介于3V~5.5V之间。

工作电流具有典型值和最大值。对某些制造商而言，典型值就是实际值。而对其他制造商而言，典型值可能就是期望值。实际值通常更接近最大值而非典型值。附注告诉我们，当芯片工作在4MHz时，此处所提供的值才成立。由于设备所使用的电流将会受到工作速度的影响，此参数的用途很有限。有些制造商将会提供公式或图表，让我们根据振荡器的速度来决定工作电流。

休眠电流表明，当此芯片进入休眠或省电模式（Power Down）时，能吸入多少电流。如你所料，它将会远少于芯片“醒着”时的工作电流（上一个参数）。不过附注指出，此休眠电流只有在振荡器的速度为32kHz时才成立，而工作电流的振荡器速度为4MHz。有些制造商会故意这么做，让他们的产品看起来比实际上还要好，因为这样做可以掩盖元器件的真实性能（别忘了，实际的电流取决于时钟速度）。这让我们很难判断一个设备在我们的应用中实际会吸取多少电流。休眠电流在时钟速度为32kHz的时候可能是10nA，而在时钟速度为4MHz时却可能是0.5mA。制造商的另外一个花招是，仅提供内核（即CPU）的工作电流，而忽略电路板上所有外部设备所用的电流。这样做表面上很好看，但是实际情况可能大相径庭。确定实情的唯一可行做法，就是把它构建成一个系统，实地去测量一下。

此例中最后一个参数是输入电容。每一个输入引脚都有一个小的电容，这与信号线（电路板上的走线）上所具有的小电阻形成一个RC电路。本章稍早部分我们就曾看到过，一个输入脉冲进入RC电路之后，会在输出端产生一个上升电压。这就意味着，尽管输入电压可以很快地从低变高，但是引脚的输入电压将会在芯片“看到”此变化之前造成些许延迟。必须在设计中将此现象纳入考量，特别当你要面对快速变化的信号时。

在这个特定的示例中，输入电容被标识为TBD，这是系统设计者的不幸。TBD代表To Be Determined（即悬而未决）。换言之，此参数在编写技术手册时还不是很明确。TBD将会出现在技术手册的任何一处，如果你正想设计一台机器，这将成为一个大问题。我还记得多年前有一个处理器技术手册，它的每个参数（从工作电压和电流到电压级别和时序）都被标上了TBD！

最后，请注意，某些（而非全部）技术手册中的技术数据只是理论值而已。换言之，芯片或元器件的设计工程师应该使用超级计算机（或算盘或塔罗纸牌）来计算这些值。有些制造厂商会颇费周折地测量它们的技术参数，甚至提供这些测试结果的详细说明。既然他们如此周详谨慎，你可以放心地使用他们的元器件。Maxim和Texas Instruments就是此类厂商的代表，尽管这样的厂商不少。

交流特性与时序

时序图（timing diagram）是设备的输入输出信号的表示方法，而且从中还能体现出输

入输出信号间的关系。实质上，它表明的是什么时候一个信号需要确认，什么时候你会得到一个期望的响应。对于两个相互作用的设备而言，它们之间的时序信号必须是兼容的，否则必须增加额外的电路以使其兼容。

由于时序图看起来的确会让很多人感到迷惑和畏惧，所以它常常被完全忽略。不过可以肯定的是，如果你忽略它，那么你设计的系统将很难工作。更何况时序图并非像看起来那样难以理解和使用。如果你想设计和构建一个可靠的系统，记住时序就是一切！

数字信号可以为以下三种状态之一：高电平、低电平或高阻态（三态）。图 4-59 所示即为用于数字设备的时序图的这三种状态。

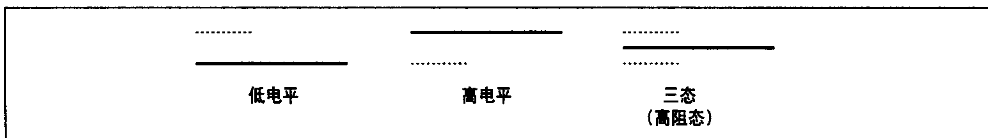


图 4-59：数字信号状态图

图 4-60 所示为一种状态到另一种状态间的转换。

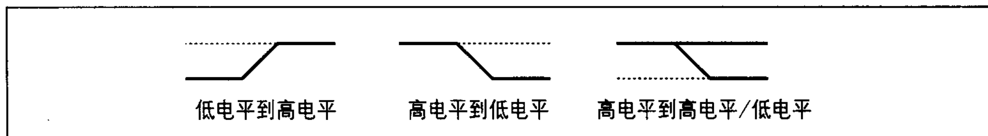


图 4-60：状态间的转换

上述波形表明如果一个信号为高电平，那么在时间轴上某个特定的点处，信号既可能保持高电平，也有可能转换为低电平。同样，一个处于高阻态的信号线，也可以转换成高电平或低电平，具体如何转换由系统的状态决定。一个这样的示例就是数据线，它在信息开始传输之前都保持三态，而一旦信息开始传输，它就变为高电平（data=1）或低电平（data=0），如图 4-61 所示。

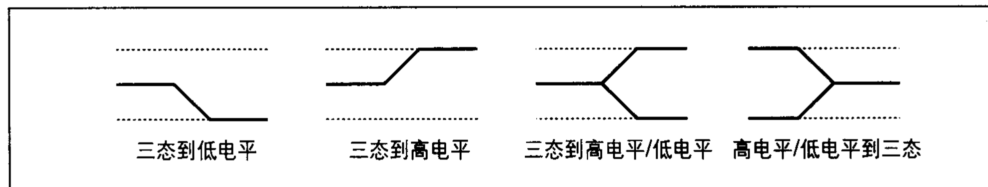


图 4-61：三态转换

图 4-62 中的波形表示从三态到高/低电平又回到三态。这表明状态的变化可以发生在给定时间段上的任何一点，但必须发生在时间轴上的指定点。

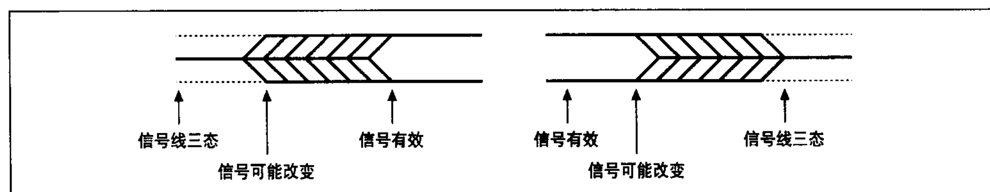


图 4-62：时序的转换

图 4-63 中的波形表明，一个信号可以/将在任意给定时间点改变状态，信号可以处于高电平并持续这一状态，或是变为低电平。或者，信号可以处于低电平并持续这一状态，或是变为高电平。

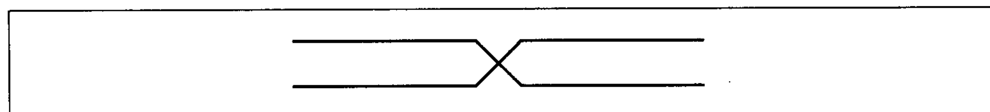


图 4-63：信号状态的改变

很多参考书都给我们这样的印象：数字电路中信号状态的改变都是瞬时发生的。其实不然。这种变化从来都不是瞬时发生的；它可能会持续几纳秒（十亿分之一秒），并且持续的时间会随设备的不同而差别很大（如图 4-64 所示）。每个元件的技术手册上都会详细列出特定设备的转换时间。

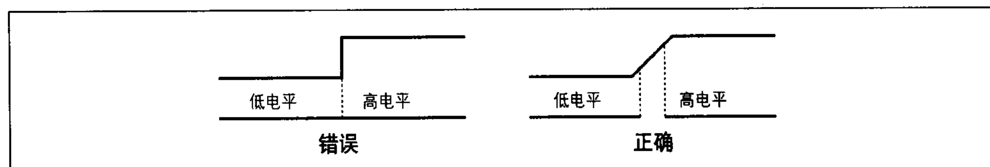


图 4-64：信号变换时序

元件制造商的技术手册中列出了所生产的设备的特定时序信息。图4-65所示为一个虚构的芯片的时序图。

时序图体现了设备的输入信号（如 $\overline{\mathbf{CS}}$）与设备的输出信号（如 Data）之间的关系。图中的编号参见时序信息列表，它们没有直接表示电路的时序。表 4-3 给出了一些技术手册中常见的时序参数信息。

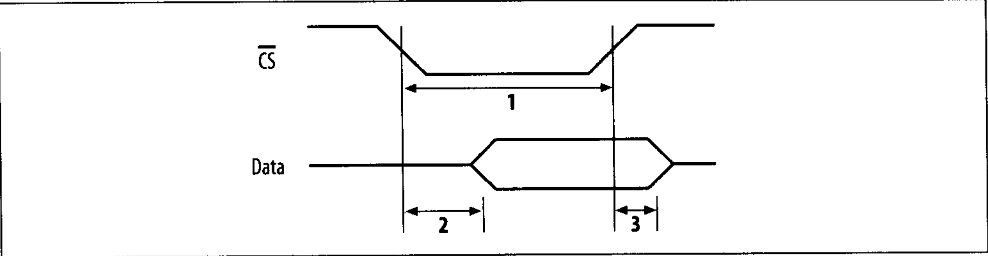


图 4-65：时序图示例

表 4-3：时序参数示例

参考	功能描述	最小值	最大值	单位
1	CS 持续时间	60		ns
2	CS 到有效数据输出		30	ns
3	Data 持续时间	5	10	ns

时序参考"1"（表 4-3 的第一行）给出了 $\overline{CS}$ 必须持续为低电平的时间。本例中，最小的持续时间为 60ns。这意味着 $\overline{CS}$ 必须持续低电平至少 60ns 才能被设备所识别。$\overline{CS}$ 的持续时间没有最大值的限制，超过最低要求的 60ns 对系统没有什么影响。

时序参考"2"给出的是当 $\overline{CS}$ 为低电平时，设备响应这种变化所需要的时间。从 $\overline{CS}$ 变为低电平直至设备输出数据的时间最多为 30ns，这意味着在 $\overline{CS}$ 变为低电平后的 30ns，设备将有效的数据输出到数据总线上。当然，输出数据也可能早于 30ns，可以确定的就是设备的响应时间不会超过 30ns。

时序参考"3"说明，一旦 $\overline{CS}$ 信号反转，设备应该在什么时候停止数据的传输。其中给出的参考值最小为 5ns，最大为 10ns。这意味着，当 $\overline{CS}$ 反转后，数据传输将持续 5ns，但不应该超过 10ns。

一些厂商使用的是时序参考表，另外一些则使用标签，如图 4-66 所示。

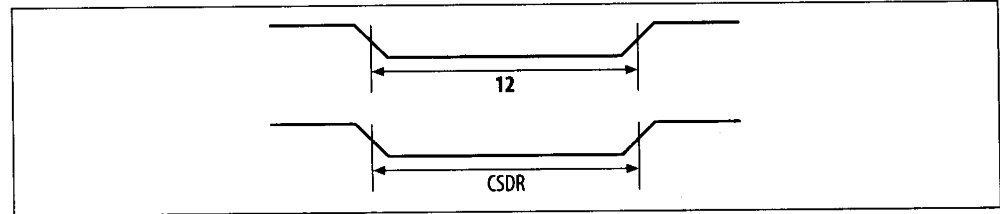


图 4-66：时序参考

一些制造商把时序长度定义为一个信号从变为有效直至变为无效的时间段。还有一些制造商则将时序长度定义为两次相邻变化中间时刻的时间间隔（如图 4-67 所示）。

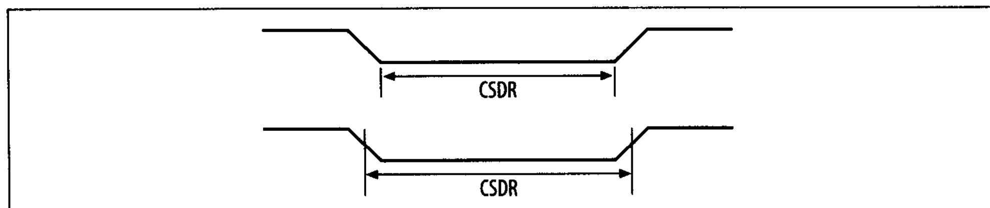
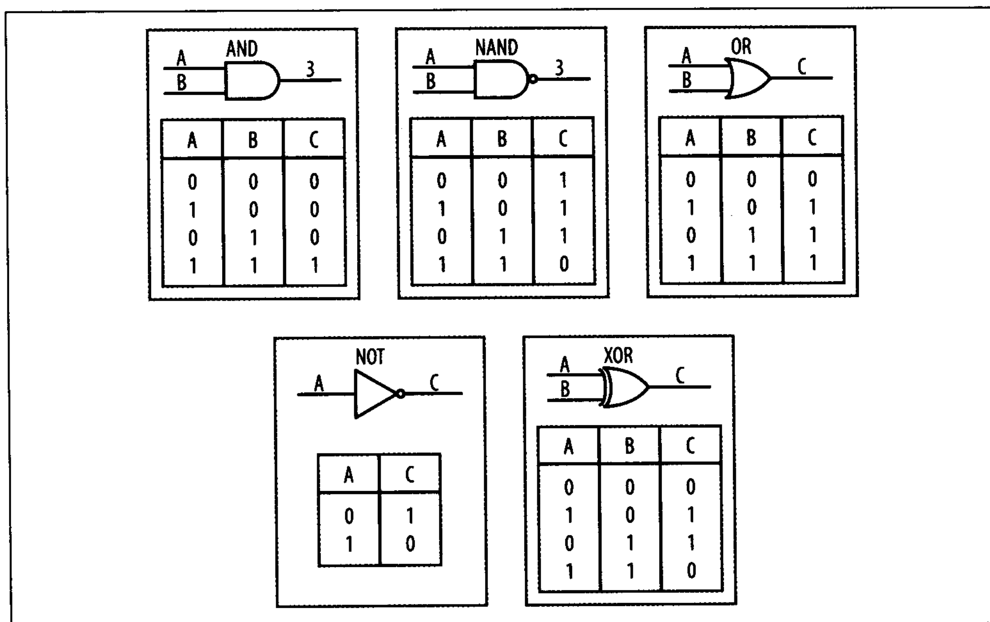


图 4-67：时序长度

逻辑门

在过去，计算机（以及其他的电子设备）都是由个别的逻辑门构建而成。如今逻辑门已经很少被用到，可编程逻辑成为了优先选择的技术手段。然而，我们仍旧可以在电路原理图中看到零星的逻辑门。图 4-68 所示为最常用的逻辑门及其真值表。其中，A 和 B 代表输入，而 C 代表输出。



AND

A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

NAND

A	B	C
0	0	1
1	0	1
0	1	1
1	1	0

OR

A	B	C
0	0	0
1	0	1
0	1	1
1	1	1

NOT

A	C
0	1
1	0

XOR

A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

图 4-68：标准逻辑门及其真值表

阅读技术手册的重要性

在进行设计之前，你需要为你的系统制定基本需求(它将做什么？其成本为多少？)，进而选择你将用到的主要器件(如选择处理器、I/O和存储器等)。在你进行其他事情之前，获取这些元件的技术手册/书籍，并从头到尾认真研读一遍。技术手册通常可以在制造商的主页上找到(本书所用的每一个芯片都是经过精心挑选的，因为每一个芯片都可从网上得到完整的技术资料)。你只需登录制造商的主页就可以下载相关文档。

一旦你彻底读完这些数据资料，觉得自己理解这些内容了，回过头再读一遍你在第一遍时所遗漏的部分内容。不要抱怨，这会对你大有裨益。把它当作一次品格塑造练习。在你设计和构建一台计算机之前来发现你所遗漏的关键内容，总比在设计和构建好之后才发现要好。

在开始设计之前，确保你得到的是最新的技术手册和勘误表（技术手册错误列表)。使用稍微有些过时的技术手册也是不明智之举。随着时间的推移，元件的电气和技术规格通常会发生改变，因此使用最新（通常是最准确的）的技术手册就格外关键了。

对你而言理解设备如何工作是非常重要的。当你调试系统时，你必须知道应该检查计算机里的哪些部件来判断部件是否正常工作。不要对设备的功能性作任何假定，认真读取并检查每一细节，包括电压标准、基本时序以及其他任何可能与系统有关的事情。

注意： 如果可能，在你的设计里包含串口是非常有用的，即使在最终应用里你不需要这个接口。串口对于从系统里打印诊断和状态信息非常有用，也是不可缺少的诊断工具(无论对于软件还是硬件)。另外一个必要的调试工具就是状态LED。如果软件和程序员聪明地使用了LED，那它就能告诉你正在测试的机器的许多信息。你所用的状态LED越多，你的调试工作就越轻松！

至此就结束了我对电子学基础知识的介绍。如果你有兴趣进一步学习，请选读与这一主题相关的更有深度的图书。电子学的确是一门令人神往而又复杂的领域，除了理解我在这里所阐述的内容之外，还有许多内容值得你更为深入地学习。既然我们讲述了电子学的基础内容，在下一章我们将学习如何为嵌入式系统提供电源。

第5章 电源

计算机要注意的广度仅如它的电线那么长。

——Turnaucka 定律

所有嵌入式计算机设计必须包含的一个重要方面便是电源。在本章中，我们将学习如何为计算机供电和稳压，以使电源平稳可靠地工作。

嵌入式计算机系统是需要电源的。在涉及到为系统供电时，有几个选择：煤、核、水、地热或电池。前四者通常归为“来自壁上插座的电流”一类。

来自壁上插座的电流

如果你的系统无需便携，来自壁上插座的电流便是最佳的一种供电方式。但从电线传出来的是交流电，且其电压很高不能立即用于数字系统。这种高压交流电必须转换为电压低得多的直流电。有很多解决办法可以实现这种转换，可以使用实验室直流电源、标准PC电源（对你的系统而言或许是大材小用）或交流电源适配器。其中，交流电源适配器对大多数应用来说可能是最好的选择。

交流电源适配器（也叫电源插头包，有时也叫作动力储存卡）是一些小的黑盒子，也可以对移动电话和其他电器供电。这种解决办法便宜、省事并且可靠，从任何不错的电子设备商那里都可以购买到交流电源适配器。它通常提供从+5VDC到 +12VDC不等的输出电压，提供的电流可以高达500mA，具体的电压和电流取决于特定插头。从中挑选一个可以为你的系统提供合适电压和电流的即可。这里要告诫你的是连接器的极性。有些交流适配器的正极电压在连接器插座的中央，而接地在外面；而另外一些恰恰与此相反！极性不明可能给你的嵌入式系统造成灾难性的后果。因此，一定要查看技术资料。

一个较好的办法是为你的设计加个桥式整流器，如图 5-1 所示。图中输入为直流，但连接的极性无关紧要。嵌入式系统使用整流器的输出作为它的电源，并且有一个内部稳压（我们很快就会讨论电源稳压）。

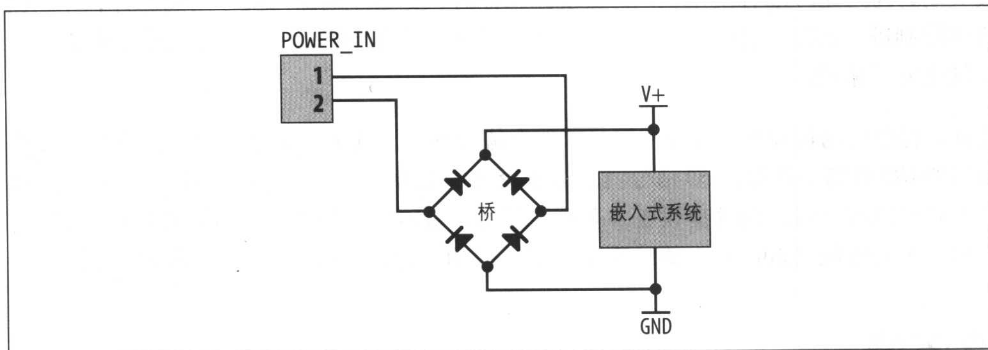


图 5-1：桥式整流器确保嵌入式系统的“供电极性”

电池

电池易于使用，唯一不足就是你所选的电池（或电池组）必须提供合适的电压以及足够的电流。只有电池选择恰当、系统设计仔细，你才能实现嵌入式系统长时间的扩展运转。例如，一个很小的基于 PIC 或 AVR 的计算机能够（依赖于应用和设计）运转两年之久，而仅仅耗费一节AA电池。而一个设计得很糟糕的系统则会在几分钟内消耗掉一节电池。电池选择不当就不能为系统提供足够的电流，会导致系统运行不稳定，甚至使系统根本就无法启动。选择电池时，不仅要考虑其平均电流量，还要考虑到其峰值电流。因为一个嵌入式计算机可能平时只需要20mA的恒定电流，但它在峰值负载时需要100mA的电流！对于使用闪存的嵌入式系统而言，这种情况尤为现实，因为闪存在写操作中需要较高的电流。这种系统的电池不仅要能够在持续负载时给系统供电，也必须能够给峰值负载时的系统供电。

低功耗设计

可以通过几种方式来降低嵌入式系统的功耗。最显著的方式是采用低功耗设备。不同设备的功耗相差甚远，许多普通的设备都有其低功耗的产品。RISC处理器的功耗通常比同类CISC处理器要低得多，因此在低功耗应用中应该优先采用RISC处理器。PIC和AVR微控制器所需电流低于5mA（在休眠模式下低达10nA），显著地小于68HC11微控制器所用的35mA。

许多存储器芯片和外围设备都会在不用时进入低功耗模式。然而，有些设备的功耗还可以进一步降低。降低设备功耗的一个非常有用的方法是在不用时就关掉它。例如，如果处理器正从RAM执行代码，并向串口输出数据，那么ROM和其他的I/O设备就可以直接关闭，因为此时没有使用它们。实现此功能时，要隔开电源与被停用的芯片，并经由软件控制进行切换。有些稳压器（下节将会讨论）具有切断输入端，可以使用它们关闭所供电的子系统。

此外，有些低功耗设备（如传感器）所需电流非常小，以至于这个电流可以直接从微控制器的I/O引脚中获得。I/O信号就是这类设备的电源。通过触发I/O引脚，在软件的控制下就可以打开或关闭这些设备。有些处理器，如PIC和AVR，就能够通过I/O引脚吸取相对大的电流（20mA），而这个电流就可以用于保证某些设备（如LED）供电。

稳压器

稳压器（voltage regulator）是一个把输入的DC电压（通常为一个输入电压的范围）转换为固定输出DC电压的半导体设备。它们主要用于在系统内提供一个恒定电压。

虽然嵌入式系统里的许多元件在一个很宽的电源范围内都可以运转，但一个固定的工作电压对这种设备也是必要的，如A/D转换器（ADC），因为许多设备使用这一内部电压来作为参考电压。换句话说，对一个传感器的输出电压的采样值是以占ADC电源电压的百分比来计算的。如果所供电压未知，那么ADC所做的任何采样都没有任何意义了（在第13章将介绍ADC及采样）。

因此，就需要一个稳压器来提供恒压电源，从而提供恒定参考电压。进一步讲，稳压器有助于去除电源的噪声，给由外部电源供电的嵌入式系统提供了一定程度的保护和隔离。如果你的系统靠电池供电运行，那么系统中变化着的电流结合电池内部阻抗的作用，将产生一个变化的电压。为嵌入式系统添加一个稳压器就可以防止这种问题的发生，在你的设计中添加稳压器是个很好的习惯。

注意： 关于使用和设计稳压电路，National Semiconductor有一个很好的在线指南，你可在 *http://www.national.com/appinfo/power/webench* 中阅读。

我们将介绍的是被称为DC-DC转换器的稳压器类型。这类稳压器接受尚未稳定的DC电压（经常为一个可能的电压范围），而供应一个固定电压值的稳定DC电压。

有三种类型的DC-DC转换器：线性稳压器（linear regulator），它产生比输入电压低的电压；开关稳压器（switching regulator），它能升高电压（升压）、降低电压（降压）或

翻转输入电压；充电泵（charge pump），它也可以升压、降压或翻转输入电压，但其电流驱动能力有限（并非所有的充电泵都提供稳定电压）。

任何稳压器的转换过程都不具有100%的效率。稳压器本身也要使用电流，即静态电流（quiescent current），这个电流来自输入电流。静态电流越大，稳压器的功耗越大（所以会发热）。在选择稳压器时，选择能为你的嵌入式系统提供适当电压和所需电流，同时它的静态电流也最低的那一款。

线性稳压器体积小、便宜、噪声低且非常易于使用。线性稳压器的基本电路如图5-2所示，其输入和输出使用退耦电容来过滤，除此之外，就不需要其他器件了。

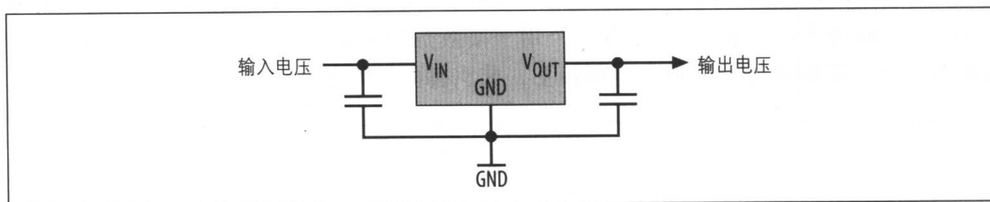


图5-2：线性稳压器示例

除了有助于平稳电压以外，电容也有利于去除电源里的瞬间短时脉冲波形干扰（叫作电压不足）。电源的这种瞬间变弱很少发生，但一旦发生就会严重地破坏计算机的运行。

注意： 许多微处理器包含电压不足检电器，一旦通过处理器的电源输入引脚的电压过低，检电器就会重启处理器。

开关稳压器由于在其输出端开关功率管MOSFET（注1）而得名。在变换输入电压为输出电压时，开关稳压器较线性稳压器更为有效；换句话说，它的功耗比较低。然而，其缺点就是需要较多的外部器件（如电感和二极管），因而占用的空间较大。开关稳压器价格也比线性稳压器高，并且产生的噪声也更多。与线性稳压器不同的是，这类稳压器能够升压、降压以及翻转电压。举例来讲，开关稳压器能够接受3.6V的电池电源，经过转换为你的嵌入式系统提供一个稳定的5V电源。或者，开关稳压器可能接受一个未调节的8V电压，转化后提供一个稳定的－12V电压。开关稳压器比线性稳压器功能强大得多。然而，它需要进行仔细的设计和电路板布线，因此请仔细注意元件生产厂商的相关说明书。

注1： MOSFET（Metal-Oxide SemiconductorField Effect Transistor）是一种高输入阻抗、低开关速度及低功耗的半导体器件。

与开关稳压器相似，充电泵能够升压、降压或者翻转输入电压。不同之处在于它不需要外部电感。然而，由于其有限的电流供应能力，充电泵很少使用。第 9 章所讨论的 MAX3222（以及类似的装置）就使用了内部充电泵来产生RS-232C移位所需的+12V 和 −12V 的电压。

LM78xx 稳压器

可以毫不夸张地讲，有几千种电压稳压器可供使用。但最常用可能是LM78xx线性稳压器系列，该系列由几个厂商来生产，如 Fairchild Semiconductor（*http://www.fairchildsemi.com*）和 ST Microelectronics（*http://www.st.com*）。它的封装格式一般为 TO-220（如图 5-3 所示），其上有一个用于接散热片的金属附接点（如图 5-4 所示）。LM78xx 线性稳压器通常平放在电路板上，其引脚向下弯折 90 度。

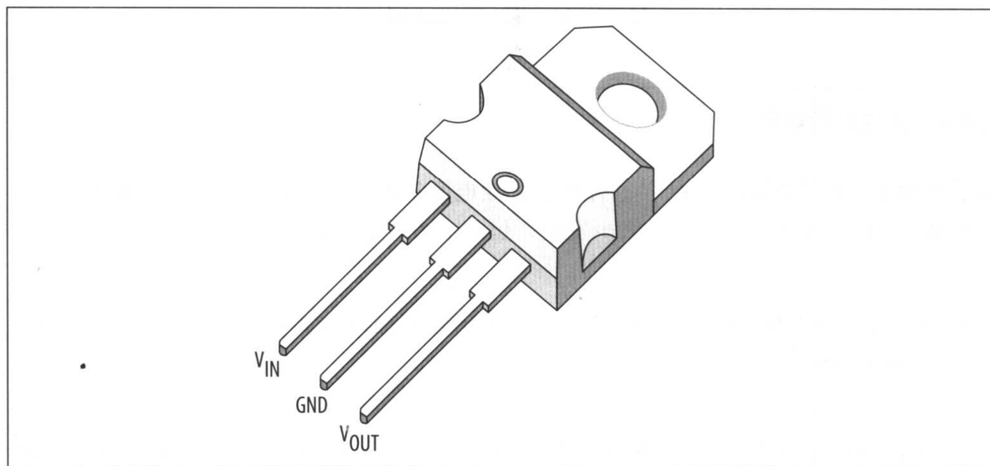


图 5-3：LM78xx 线性稳压器

LM78xx 的 xx 即表明了输出电压。例如，LM7805 提供了 5V 的稳压输出，而 LM7812 提供了 12V 的输出电压。这种器件的输出电流高达 1A（峰值达 2.2A），而静态电流在 5mA~8mA之间。LM78xx元件的另外一个特性是超载和短路保护。表5-1列出了LM78xx 稳压器系列及相应的输入电压范围和输出电压。

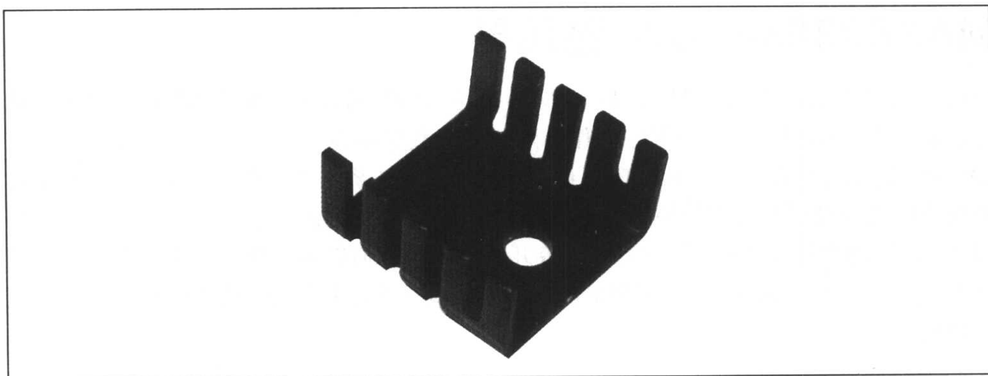


图 5-4：散热片

表 5-1：LM78xx 系列稳压器

型号	输出电压（V）	输入电压范围（V）
LM7805	5	7~25
LM7806	6	8~25
LM7808	8	10.5~25
LM7809	9	11.5~25
LM7810	10	12.5~25
LM7812	12	14.5~30
LM7815	15	17.5~30
LM7818	18	21~33
LM7824	24	27~38

LM78xx 系列稳压器使用简单，在输入（引脚 1）和输出（引脚 2）上需要退耦电容（通常在 10μF~47μF 之间），引脚 2 接地，如图 5-5 所示。

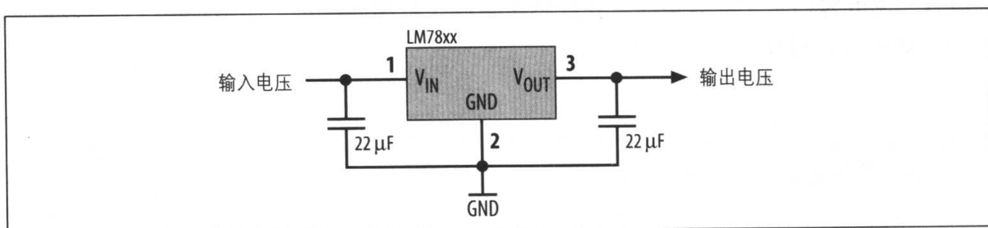


图 5-5：LM78xx 电路

如果要输出负电压，就用 LM79xx 稳压器，其用法和 LM78xx 极为相似。

MAX603/MAX604 稳压器

Maxim 公司的MAX603（5V 输出）或MAX604（3.3V 输出）都是很不错的稳压器。我常在我的许多小型设计中采用它们作为默认重负载稳压器。这种稳压器的静态电流较LM78xx稳压器小得多，这对低功耗或基于电池运转的系统非常理想。这些开关稳压器的封装格式为极小的贴片封装SO-8或标准DIP（第6章讨论），它们只需两个外部元件即可。机敏的你可能会问，仅需退耦电容开关稳压器如何能够工作！电感和二极管到哪里去了？答案是，Maxim已经将所需的东西都放到一个芯片里了，这就使得设计工作更为简易。

MAX603/604能够提供高达500mA的电流，能够在+2.7V ~ +11.5V 的输入DC之间工作。它们有内置保护，以防你不经意地交换了电源和接地，其电流消耗也只有15mA。因此，把它们用于低功耗嵌入式计算机里是很理想的选择。稳压器如MAX603或MAX604的使用示意图如图5-6所示。在本例中，输入电源用的是电池，DC插头或者其他类型的电源也是同样的简单。

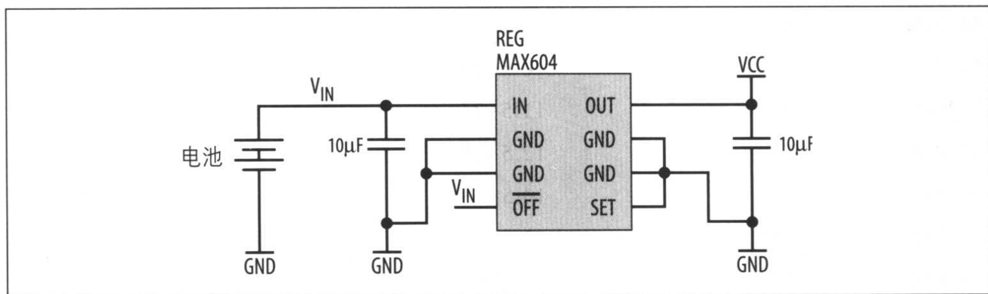


图 5-6：MAX603/MAX604 电路

在输出端（稳压器的引脚8）需要一个电容来过滤电压。这个电容也便组成了稳压电路的一部分，没有它稳压器就无法调整电压。第二个所需的支持元件是输入端（引脚1）上的电容。该电容稳定了输入电压，并能够在峰值负载期间提供一个附加电流源，我们称其为退耦电容，对此本章稍后将详细讨论。

注意： 为每个电源引脚供电的任何元件都应该有一个退耦电容。这点很重要，如果忽略这点，你的计算机工作起来必定不可靠。

接地引脚（2、3、5、6和7引脚）应该都接地，因为它们可以用作设备的小散热片。在进行PCB布线时，稳压器的下面放上接地覆铜，并将上述5个接地引脚都直接连到接地

覆铜上。引脚 4（$\overline{\textbf{OFF}}$）把电容置为关闭模式。对于持续操作，这个引脚需要直接连接到电源，以使设备一直处于打开状态。如果你使用该电容来控制你的嵌入式计算机中的子系统，可以通过将 $\overline{\textbf{OFF}}$ 连接到处理器的 I/O 行来使得 $\overline{\textbf{OFF}}$ 作为软控制的电源开关。

MAX604 提供了 3.3V 的稳定输出电压。要得到 5V 的输出电压，把 MAX604 替换为 MAX603 即可，电路的其余部分完全一样。

MAX1615 稳压器

MAX1615是一个线性稳压器，在4V~28V电压之间都能工作。它是一个小巧的元件，在3.3V 或 5V 下可以提供 30 mA 的输出电流。目前，30mA 不算是大电流，但对于非常小（电池供电）的应用，这个电流已经足够了。MAX1615 有一个关闭输入引脚，允许外部系统通过这一引脚来关闭它。这种稳压器的用途之一就是作为嵌入式计算机内部子系统的电源，在不使用这些子系统时，就允许处理器关闭它们。在关闭一个系统之前，一定要确保它的“缺席”时不会影响到系统的其余子系统的功能。

MAX1615 电路的基本原理图如图 5-7 所示。

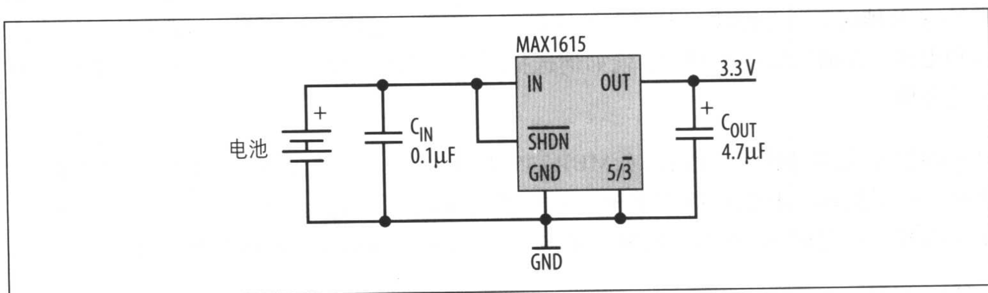


图 5-7：MAX1615 电路

引脚5/$\overline{3}$决定了输出电压，通过将其接地，输出电压就是3.3V；把此引脚连到输入电压，输出电压就是 5V。要使系统持续运转，把关闭引脚（$\overline{\textbf{SHDN}}$）接到输入电压即可。如果要关闭稳压器，可以把 $\overline{\textbf{SHDN}}$ 接到处理器 I/O 线路或接到一个简单的电源开关（毫无疑问，如果你用处理器的 I/O 线路来驱动 $\overline{\textbf{SHDN}}$，那么处理器就必须有一个除了这个稳压器之外的电源！否则，就会导致一些有趣的情形发生）。

MAX724 稳压器

对于高电流的应用情况，以 8V~ 40V 之间的输入电压，MAX724 能够提供高达 5A 的电

流，而它的输出电压在 2.5V ~35V 之间可调，静态电流是 8.5mA。这个芯片是 5 引脚 TO-220 封装，带有一个外部散热片附接点。MAX724 的基本电路如图 5-8 所示。注意，这是一个只降压的稳压器。

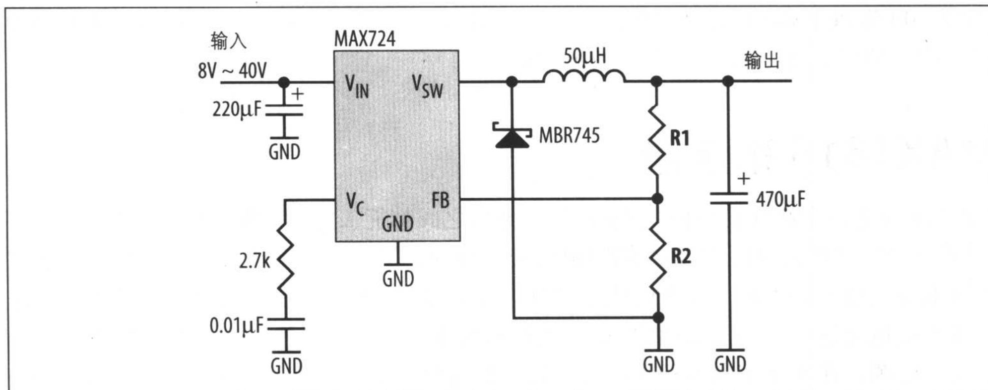


图 5-8：MAX724 电路

图 5-8 中的电感标称值是 50uH，但实际上可以是 5μH~200μH 之间的任意值。当输出引脚 V_{SW} 关掉时，二极管就为电感电流提供了一条接地通路。所选择的电感应该有一个高饱和电流，否则效果会很糟糕。就如何选择适当的电感，Maxim 的技术手册上提供了详细的说明。

Maxim推荐使用肖特基二极管，如MBR745，因为这种二极管符合切换时间很快的要求。这种稳压器的输入和输出都需要较大的退耦电容来进行滤波。但所选的电容在预期的温度范围和运行时间内，必须有较低的等效串联电阻（Equivalent Series Resistance，ESR）。

其输出电压由 R1 和 R2 来调整。计算输出电压的公式是：

$$V_{OUT} = 2.21 * (R1 + R2) / R2$$

R2 的阻值应该小于 4kΩ，Maxim 推荐值为 2.21kΩ，这样上一公式就简化为：

$$V_{OUT} = (R1 + 2.21k) / 1000$$

因此，计算 R1 的公式为：

$$R1 = 1000 * V_{OUT} - 2.21k$$

到目前为止，在大多数情形下我所介绍的稳压器都应该是可用的。如果你在设计一个有特殊需求的机器，一定要选择一个适合你的需求的稳压器。花些时间去浏览一下元件制造商的主页，看看他们可以提供何种性能的稳压器。

电气噪声与干扰

本质上，数字系统运行的是模拟信号。由于系统里模拟信号的作用，数字信号会产生衰减并会受到噪声影响。来自附近电气设备或无线电转播中的寄生噪声和反射波会在你的电路之内感应出信号，这一信号能够导致错误事件的发生，甚至能够使数字系统的功能完全丧失。确保你的系统免受电磁干扰的方法之一是避免使用电源！很不幸，蒸汽动力的微处理器是不现实的，因此如果你的系统要在现实世界里可靠地运行，就不得不考虑电磁的影响。下面我们所讲的内容，不是关于噪声及其相关问题和解决方案的一个综合概述。这是一个异常复杂的领域，在此远不能完全地讲述。我所要做的只是提供一个该领域的入门。噪声这一学科值得去搜寻更为详细的信息，值得更透彻地去理解这些概念。

警告： 美国、澳大利亚、欧盟以及其他许多国家对电磁辐射和抗扰度都有严格的指导和要求。我在这里所提出的一些建议只是很好的经验，但不见得能够符合你所在国家的许可。因此，你要查看一下当地的一些规定，确保所作的设计符合要求。仅通过设计是无法保证符合电磁要求的，还必须对系统进行测试。

噪声是数字系统里的一个重要问题。噪声可能干扰信号传输，从而破坏数据，或者导致程序崩溃。嵌入式系统所发生的问题可能与噪声相关，也可能与噪声无关。例如，电源供电不足、退耦电容不够、边际时序冗余（marginal timing tolerances）以及软件 bug 都可能在运行时导致随机的短时脉冲波形干扰。然而，即便是构思巧妙的系统也可能被噪声所毁坏掉。设计中的噪声问题是固有的。来自集成电路的开关噪声、铃声以及话音串扰都是设计嵌入式系统所要考虑的问题。其他形式的噪音可能是因环境影响而起（例如附近电机或无线电辐射等）。麻烦的是，对数字系统而言，电磁问题越来越严重了。数字系统被来自无线电、TV 和移动电话发射塔的不断增加的辐射所包围着。同时，由于设计朝着高速低功耗发展，集成电路变得日益敏感。

上述噪声源在设计和测试期间可能不会出现，但一旦系统投入使用，那些问题便会出现。嵌入式系统垮掉可能是因为硬件问题、软件问题或者在一个街区以外的工厂打开了压缩器（译注 1），从而在供电上产生一个峰值信号。由噪声所产生的问题可能偶尔发生，但常常很难对其进行跟踪。有些问题常常每隔几天就会出现。任何问题都是令人不满的，都必须加以解决，但是确认问题的原因却总是不那么容易。最好从一开始设计时就使系统尽可能地免受这些问题的困扰。你必须考虑的不仅仅是你的系统可能产生的电磁辐射，还有外部作用对系统的影响。这并不能保证你的系统不会有任何问题，但尽量多地考虑到可能的问题对系统是有帮助的。

译注 1： 减小某特定范围内强弱信号之间的差别的一种电路。

电磁干扰（ElectroMagnetic Interference，EMI）是由嵌入式系统之外的信号源产生的。EMI的一些例子有电机、功耗开关、荧光照明灯、RF辐射以及静电释放等。所有这些都可能是重大的噪声源。例如，打开或关闭电动马达会在AC电源线上产生一个1000V的峰值电压。静电释放（Electro-Static Discharge，ESD）能够以每秒4A的电流增长速率从手指发送一个峰值为35kV的电压到集成电路上！这足够永久性地损坏一个敏感芯片了。汽车是特别的噪声环境，汽车电子设备的12V供电线可能翻转，而以从6V到高达24V的电压驱动设备，其瞬时电压峰值可达400V。所有这些都会对一个嵌入式系统的运行产生极坏的影响。

在任何电路里，都有一个电流输入线和电流输出线。通过电线的电流会在电线周围产生一个磁场，这个磁场就是EMI源。这个磁场的强度与离磁场源的距离成反比，磁场的方向与电线里电流的方向直接相关。

最小化电流环路面积

经由电源和信号线路，再通过接地回到电源（从而完成这个电路），电流便穿越了一个系统。由此，接地便构成了系统内电流的返回通路。如果信号线路与返回线路所处的位置靠得很紧密，那么线路里的电流产生的磁场在线路短距离内可相互抵消，这就是所谓的最小化电流环路面积（Minimizing the Current Loop Area），其目标就是使系统内所有的信号线路与返回线路尽可能地近并尽可能地短。在许多电流环路的系统里（这在许多大规模高速数字系统里比较常见），就用接地面来最小化环路面积。接地面是一个很大的传导表面，它能用作电路里所有环路的电流的返回通路。接地面常常作为一个完整的内部PCB层来实现。

电容耦合是电场的耦合。一条线路上的信号，通过它的相关电场，能够在一个相邻信号线里电容性地导致一个幻影“信号”。这就是所谓的数字系统的串扰。如果系统设计不正确，串扰量级可能会很大，很容易引起系统崩溃。

系统可以通过静电或法拉第屏蔽（Faraday Shield）来屏蔽信号线以降低电容耦合。这个屏蔽是置于电容耦合元件之间的一个金属导体。屏蔽是PCB的简单部分，它以与电路轨道同样的方式构成，通常是接地的（尽管不总是如此）。这个屏蔽也可能是位于集成电路下的一个简单接地面，这样安排是为了防止它受电路板内面上的信号线的影响。通过在信号线之间放置接地线，可以实现信号线之间的屏蔽（如果必要）。

保持电源平稳

与信号线一样，保持电流环路最小的原理也同样地适用系统内的电源线。然而，保持电源线的环路面积最小是很困难的。电源必须分布于整个电路，而要在整个PCB里进行有

效的电源布线，并且还要使环路面积尽可能小，就非常困难了。所以电源线易受噪声影响，这可能会对电路造成很大的问题。

解决的办法是，对电源中出现的任何噪声都提供一个接地通路。可以对电路里的每一元件局部地提供接地通路，这是通过为每一个集成电路在电源和接地之间增添一个退耦电容来实现的。这个电容对来自电源的噪声进行退耦，为噪声提供了一个到达接地的通路。通过这种方式，噪声便可从电源中去除，芯片也就有了持续和稳定的电压源。这里，退耦电容应该尽可能地接近设备的电源引脚。贴片封装电容的连接具有非常低的自感应，因此更为可取。陶瓷电容由于其低阻抗性也常用作退耦电容。

电容还有另外的优点：当设备必须转换其输出或内部状态时，它可以用作设备的电流源。就这点而论，电容是表现极小的环路面积的电流源。一般地，通过在电源输入附近放置一个大的（比如22~100μF）电解电容或钽电容就可以对电路板进行退耦，通过为每一集成电路连接10nF的陶瓷电容就可以分别地对其退耦。使用多个退耦电容，即为每一电源引脚配置一个电容，可以获得更好的退耦效果。你有必要确保所有可能影响电路的所有频率信号都有一个低阻抗的接地通路。为此，可以选用的电容（100nF、10nF和100pF）有好几种，用以对宽范围频率的电源进行退耦，从而从电源电路里去除噪声。另外，一个板载电压稳压器能够在你的电路和外部电源之间提供一定程度的隔离。在电路里使用的退耦电容是多多益善（当然也要合情合理）。

退耦电容的重要性

许多年以前，当我在一所大学里任教时，我带毕业班的一个学生承担了一个项目，使用现在早已淘汰的Motorola 88110 MIMD处理器来设计和搭建一个64位工作站。但结果这个机器无法工作，原因在于这个学生在设计中忽略了退耦电容。当我指出这一错误并加以改正后，这台机器便运行起来了。这些简单的电容就把一台除了复位之外别无其他操作的机器变成了一台可以运行的64位计算机。

接地线中也可能存在噪声。并非各处的接地线的电压都相同。在电路不同部分的接地线之间可能存在一个电压差，虽然这些不同的接地线都“接地”。这个电压差能驱动几个安培的电流通过接地线，这被称为接地环路，它可能导致非常严重的问题。屏蔽、退耦以及最小化电流环路面积有助于保护系统免受接地噪声的影响。

当同一设备上的一个或多个输出由高到低变换时，就会在信号线上导致接地噪声（ground bounce）的振荡（振动）。这个振荡的振幅可能很大，会对系统产生严重的影响。有些设备的设计特意使接地噪声最小化。以较短的引线采用封装形式的设备（如

PLCC封装和贴片封装）能够减少振荡影响。带有朝向封装中心的接地和电源引脚的设备具有较短的引线长度，这也有助于减少振荡影响。端接法技术也有助于减少接地噪声。

你可能遇到的另外一个影响是由几个输出瞬间同时转换引起的。如同接地反跳，这个影响也来自于与芯片的封装和内部连线相关的寄生影响。当几个输出同时变换时，这些影响会给输出信号的变换造成几纳秒的延迟。因此你必须考虑到变换的输出可能对你的电路所产生的影响。

如何毫不费力地摧毁一个计算机

如果你在干燥的日子里走过地毯，或者在塑料表面上摩擦一块毛皮，静电便会产生。如果接着你或者毛皮接触金属物品，你所感到的麻酥酥的感觉便是静电释放（ESD）。ESD可以永久性地破坏掉一块集成电路。虽然静电释放可能小到无法察觉，但它依然可以轻松地摧毁一块半导体。

许多集成电路都有内部静电保护措施。这种措施足以保护设备免受在正常操作期间累积的静电的危害。然而，对于有时可能发生的巨大的静电瞬态放电，这种保护措施是不够的。一旦设备置于电路中，你就不能认定它是安全的。当打字员把指头放到键盘上时释放出来的瞬态静电就有可能毁掉一个处理器；因为这个瞬态放电如同雷击一样，会试图寻找接地的通路，即使这一通路是沿着一条数据线并穿过处理器才到达接地。

为此，一个解决的办法是，使用一个缓冲芯片把系统里的重要部件从可能与ESD接触的部件中隔离开来。这不是一个理想的解决方案，仅仅意味着会受到损坏的是缓冲芯片而不是处理器。你得到的仍旧是一个不成功的系统。

于是，能够提供保护的瞬时干扰抑制器就很有用了。这种抑制器在正常电压水平时充当开环电路，而在更高电压时将电源引导到地面。决不能用信号地来把ESD接地。大多数集成电路对于ESD把它们的接地引脚提升至高于电源几百伏都不能作出很好的反应！

在处理芯片时，使用接地垫是个不错的主意。接地垫是一个传导薄片，你可以把它连接到一个就近电源的接地。可以在你的胳膊上缠上接地带，把你自己与接地垫连接。这样一来，任何静电电荷都消散了，而且不可能有任何集聚的机会。

警告： 如果你在使用一个接地垫，那么不要忘了在加电之前把你的嵌入式系统从接地垫移开。接地垫是传导的，在接地垫连于系统时对系统加电，会造成重大灾难。

在下一章，我们将会探讨设计和构建一台物理计算机的过程。我们将会看到你应对所设计系统的运行电气环境作何种考量，以及如何进行印制电路板的布局。

第6章

搭建硬件平台

硬件，就是计算机上可以让你踢一脚的地方。

——佚名

在进入与设计有关的章节之前，让我们先花些时间来看看如何制造一台具体的机器。搭建一个不能工作的机器是非常容易的。你可能拥有完美的设计和无瑕疵的代码，但如果忽略了机器运行时所处的具体环境，那么你所构造的不过是一个蹩脚的镇纸而已。

在本章里，我将向你展示如何布置电路板（以及在哪些地方需要特别注意）以及如何调试硬件。特别地，我将具体分析如何制作ATtiny15计算机的设计（这一计算机会在第15章讲到）。我假定你只是手工制作少量的嵌入式系统，因此我的讨论也是针对这种设计展开的。我在这里所讲的内容，不是电路板设计或装配的最高发展水平，而是家庭手工业级的计算机产品的设计指导方针。如果你需要大批量地生产，那么你得已经知道如何做（那就跳过本章好了），或者要去咨询专业人员。

工具

要设计和构建一台嵌入式计算机，必须使用正确的工具。设计一台计算机首先需要的是另一台计算机。这其实是一个鸡生蛋蛋生鸡的问题。你将需要一些软件工具来进行（电路图与电路板布局的）设计，你还需要一些软件工具来编写程序，并将程序下载到你的嵌入式目标机器上。

开发套件

当你准备开发嵌入式系统时，最好将从处理器制造商那里获取开发套件（development kit）作为开始，如图6-1所示。

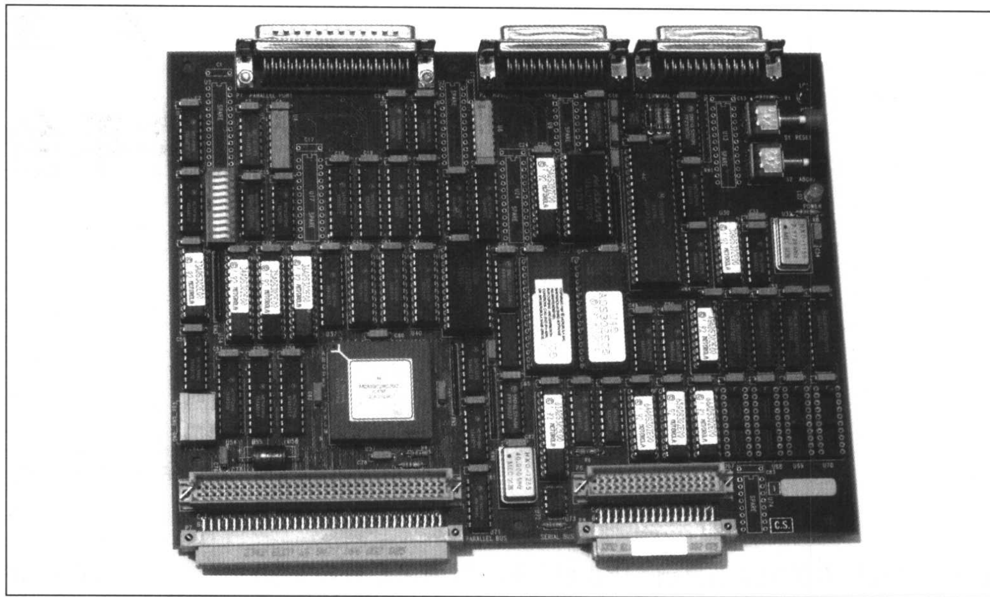


图 6-1：微处理器开发系统

一个好的开发套件不仅会为你提供你试图构建的机器的样机(在这个样机上可以测试你的代码)，它还会提供一个细致的集成开发环境（Integrated Development Environment，IDE)。IDE中有一个窗口编辑器、一个调试器，如果你够幸运还会有一个仿真器、一个汇编器，或者还会有一个C语言编译器。开发套件中还应该附带电缆、下载代码到处理器的工具以及电路原理图，通过电路原理图你可以看到运行着的机器是什么样子。请对这里的电路原理图稍加注意，有些（并非全部）半导体厂商会把它们的开发系统的设计工作承包给一些小的外包公司。在这些公司中，有一些的确作出了非同寻常的工作，而其他一些公司，看起来好像雇佣了迷路的非洲黑猩猩作为它们的设计工程师一样，所设计的产品令客户苦不堪言。后者所设计的产品若想运行起来，只有祈祷奇迹和数字上帝的慈悲才行。所以这些电路图看看就好，不应该作为一个系统应该如何设计的最佳典范。

为了使用集成开发环境IDE，你将需要一台台式计算机。这可能是一个坏消息，因为几乎毫无例外地，这些集成开发环境只能在一种操作系统和一种平台上运行。你没有任何必要去猜测是哪一个操作系统和哪一个平台，因为没有人给你发奖金。因此，如果你首选的环境是Unix工作站，通常你的运气就比较差了。虽然GNU工具很伟大，但有时你不得不把你的IDE源代码下载到你的目标计算机上，尤其是对于8位和16位处理器。

这种开发套件的价格不一，有免费的（如果你在恰当的地点恰当的时间里），而对于一些真正的高端和优良的处理器，其价格则高达好几万美金。对于大多数嵌入式类型的处

理器的开发套件，可能要花费 50~300 美金不等，这取决于所用的芯片、制造厂商及其当前的流行程度。开发套件为你所节省的时间将使你的投资很值得。

测量工具

对于本书所描述的系统，你所需的最起码的调试工具是一个万用表（如图 6-2 所示）和一个示波器（如图 6-3 所示）。



图 6-2：Fluke 数字万用表

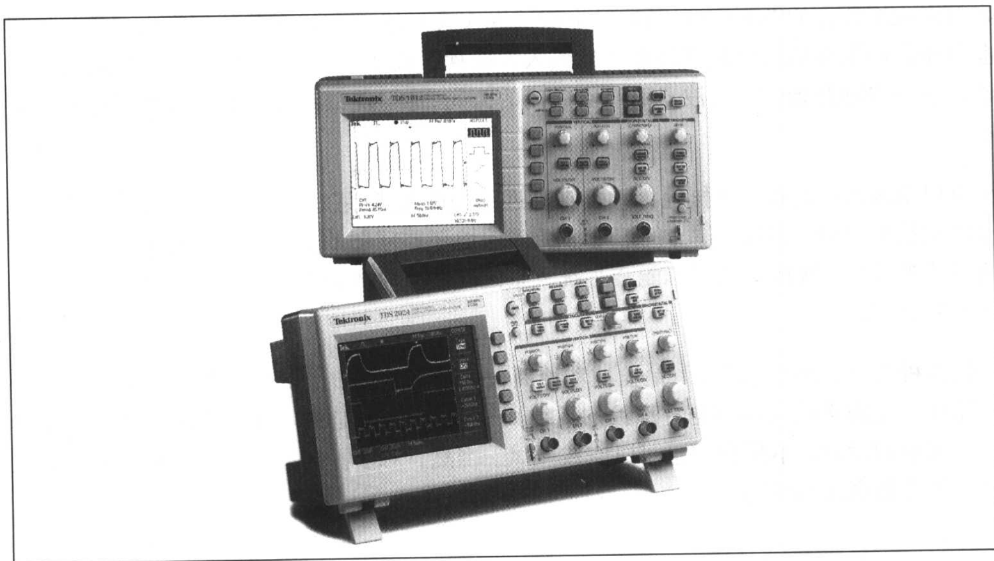


图 6-3：Tektronix 示波器

万用表够测量电流和电压，但更为重要的是，它也能够测试两点是否连续（能够证实物理上的及因此导致的电气上的连接）。但是如果你的系统里有灵敏的元件，就不要做这种连续的测试，这种测试可能会损毁它们。

警告：不要假定仅因为在走线（trace）的一端有信号，那么沿着该走线的所有点都会有信号。要用示波器探棒检测每一个地方，并且用万用表确保每一条信号通路都连接好了。

示波器可以用来观察你的系统中的波形，同样地，它也是你最基本的调试工具。示波器从简陋粗糙的到昂贵复杂的都有。虽然你不需要在示波器上花费100 000美元，但是你需要一台能够精确显示波形的示波器。所以，你从Gorsky先生的路边旧货铺里花20美元捡来的“老古董”恐怕是不能胜任的。

你需要一台足够带宽的示波器来观察你的计算机里的信号，用一台20MHz的示波器去测试一个100MHz的系统时钟是没用的。示波器根本捕捉不到信号，因此你也就不能观察到系统的波形。示波器的带宽越宽，你能观察到的信息也就越多。虽然你可能认为一个4MHz的嵌入式处理器不需要一台100MHz的示波器，但是这台示波器可以让你看到真正的波形上升沿（而不仅仅是垂直跃变），并且让你观察到极其微小的时序差异，而这些差异是可能有反作用的；它也能让你看到噪声峰值或信号线上的振荡，这些都有可能对你的机器有不良影响。

我很喜欢用便宜的Tektronix示波器来调试嵌入式系统，而HP和其他厂商生产的示波器也很棒。如果你确实很认真地想开发嵌入式硬件，投资买一台这样的示波器是值得的。请留意一些刚开始生产示波器的濒临破产的公司，你可能以低廉的价格买到不错的示波器！

在使用示波器的时候，用接地探棒接到靠近你的嵌入式系统（或者干脆就放在上面）的接地端是必要的。否则，信号的测量会受到接地环路问题的影响，从而就不能读取一个精确的测量值。这样一来你可能会花费大量的时间去做无意义的工作，而真正的问题反倒被忽略了。

逻辑分析器（如图6-4所示）是监控和诊断数字信号的工具，其价格非常昂贵。对设计高速和复杂的系统（特别是有总线的系统），逻辑分析器是最基本的调试工具，但是对于本书中的设计，不用它们也可以完成调试。当然，对于一个完全自持的微控制器，逻辑分析器是毫无用处的。

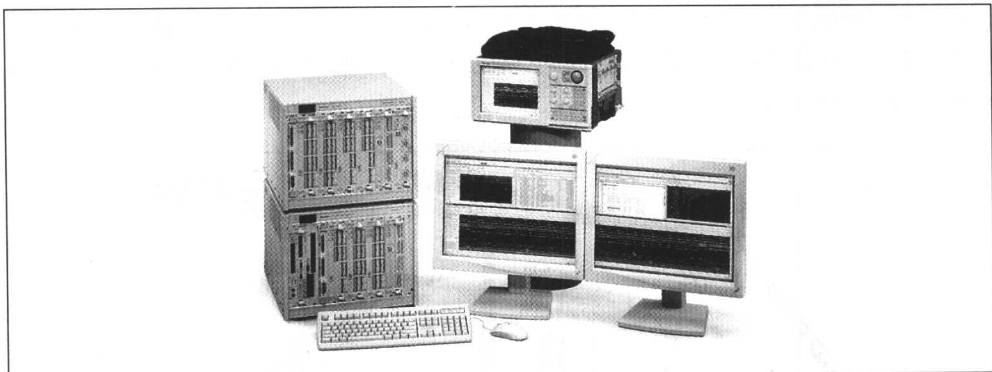


图6-4：Tektronix逻辑分析仪

在线仿真器

另外一个开发工具是在线仿真器（In-Circuit Emulator，ICE），这是一个与处理器带有相同引脚的小模块，在设计阶段把它放到目标系统中去。在运行于PC的软件的控制下，ICE模拟嵌入式处理器的运行，看起来就像一个在电路中运行的处理器一样。这样你就能够交互式地调试硬件和软件。这在基于自持微控制器的系统里是极其有用的，不用ICE而想深入到这种系统内部是极其困难的（其实是不可能的）。

有些ICE比其他的性能要好，这就是所谓的一分价钱一分货了。真正处理器时序和电气特性紧密匹配的复杂ICE是很昂贵的。廉价的ICE也能模拟处理器的操作，但却是以完全不同的信号时序来完成的。而且，对于你开发的系统里的每一种型号的处理器，都需要一个不同的ICE。

有些工程师在其嵌入式系统的开发过程中过分地倚重ICE。你不妨把我称为一个异端者，因为我不用ICE也能进行调试。ICE的缺憾就是，无论它是多么好、多么独特的工具，它都是在仿真，和真实的处理器不可能完全一样，它总会在电气特性或时钟特性上存在一些轻微的差别。波音公司的工程师有一句名言："测试你的飞机；飞机却不是你所测试的。"换句话说，真实事物是没有替代品的。

构建工具

最后，你需要一个好的烙铁、一些斜口钳、一些尖嘴钳，以及其他构建工具。切勿使用用来架设篱笆的钳子；请使用钳口非常细的高品质钳子。我有六把钳子，它们都非常适合用来做精细的工作，而且每把钳子都略有不同。我还有钟表师父所使用的一组工具，

非常适合在精细以及需要小心装配的工作上。可以的话，一台优质的显微镜不仅有助于构建工作，还有利于对最后装配完成的电路进行验证。图 6-5 所示为构建硬件平台期间我常用到的一些工具。



图 6-5：构建工具

焊接

焊接很容易做好，但也很容易做坏，它的基本技巧很容易掌握，而且成为一个焊接高手也并不困难。

警告：安全第一

要知道焊锡是含铅的，因此焊接应在通风良好的工作场所进行；焊接过程要避免吸入浓烟；焊接操作后要洗手，尤其在饭前更要注意！

焊锡容易飞溅，所以在工作时要穿着防护服和戴防护面具。

焊锡是一种熔点相对较低的金属合金，用它来将元件连接到电路板上去并形成通路。焊锡的成品有焊丝（就像一卷线）和焊锡膏（由注射器射出）。焊接工具基本有两类：标准电烙铁和拆焊台。Weller（*http://www.cooperhandtools.com/brands/weller*）和 Hakko（*http://www.hakko.com*）所生产的焊接工具都不错。在此要感谢 Weller 提供产品的照片给我，因此你可以在本书中看到烙铁和拆焊台的样子。

烙铁（如图6-6所示）和焊丝可以用来安装过孔元件。注意，海绵主要是为了清洁烙铁头之用。海绵应该随时维持非常潮湿的状态，而且你应该时常擦拭烙铁头（不要像用抹布那样将海绵挤干。海绵需要维持很湿的状态，但也不要湿透）。



图6-6：Weller WES51烙铁

有各种各样的烙铁头，这取决于你所进行的焊接工作，以及希望烙铁达到的温度。当烙铁冷却时，你可以轻松地进行更换。烙铁头被磨损后应该换上新的。备用件可轻易地从大部分电子供应商处购买到。

警告： 请不要在烙铁通电的时候拿掉烙铁头。烙铁头里包含了一个用来控温的传感器。拿掉烙铁头会使烙铁过热，从而导致烙铁损坏。

也不要在长时间不用的时候让烙铁通着电，这么做会缩短烙铁的使用寿命。

始终为烙铁头包上一层薄薄的焊锡，这么做可以避免氧化以及延长烙铁头的使用寿命。进行焊接时，烙铁头将会失去光泽。你可以在烙铁头还热的时候弄上一些焊丝，并用海绵抹掉多余的焊锡。这么做可以确保你的烙铁头被维持在不错的状态。细心照料烙铁头所得的回报是，你可以轻易完成焊接工作。对烙铁头不加维护的恶果是，你要完成焊接工作会比登天还难。

拆焊台（如图6-7所示）通过一个小的喷嘴吹出热风，它基本上是以焊锡膏来焊接表面贴片元件。然而，某些表面贴片元件使用普通电烙铁来焊接可能会比较简单。你并不一定要有很昂贵的拆焊台，尽管大部分表面贴片元件用拆焊台来焊接比较容易。无论你是否购买了拆焊台，传统的烙铁仍然是不可或缺的。有许多元件（如通孔连接器），就必须使用电烙铁来焊接。



图6-7：一个简单的Weller WAD101拆焊台

把标准焊接工具和热风枪整合在一起的集成工作台已经有了。图6-8所示为WRS3000ST套件。

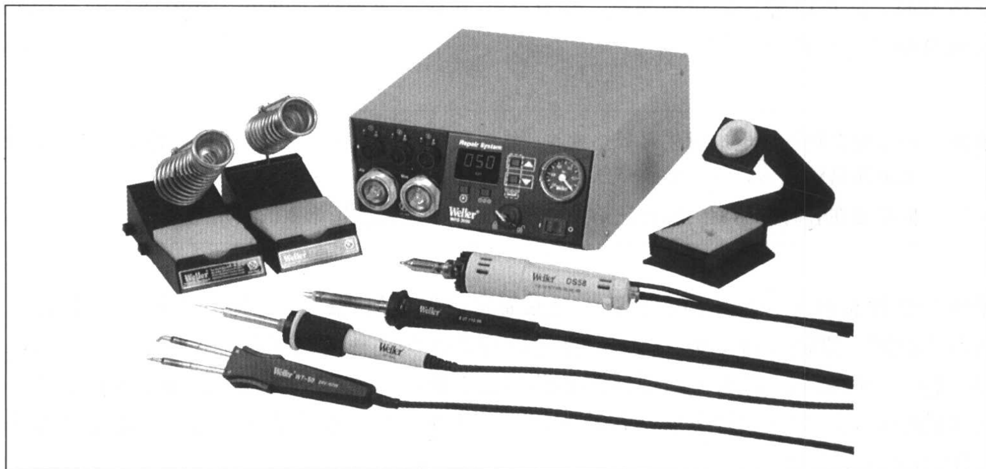


图6-8：Weller WRS3000ST

要焊接球栅阵列（ball-grid array，BGA）封装的芯片，必须使用类似Weller WQB 3000（如图6-9所示）之类的拆焊台。此类工具让你得以精确地放置元件，并成功地将数百个锡球（焊点）粘在BGA底下。如果你买不起这样的工具，可以找到为客户代为安装电子器件的公司。可以花点钱请他们帮你把元件安装到你的电路板上。

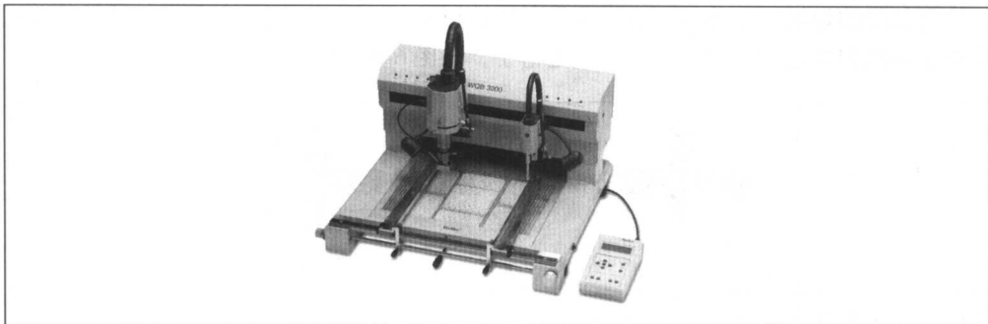


图6-9：Weller WQB 3000

焊锡所产生的烟雾有毒性，长期暴露在此烟雾下有害健康。进行焊接工作时，请注意工作环境的空气流通。为了身体的健康，买一台排烟机是值得的，如图6-10所示。

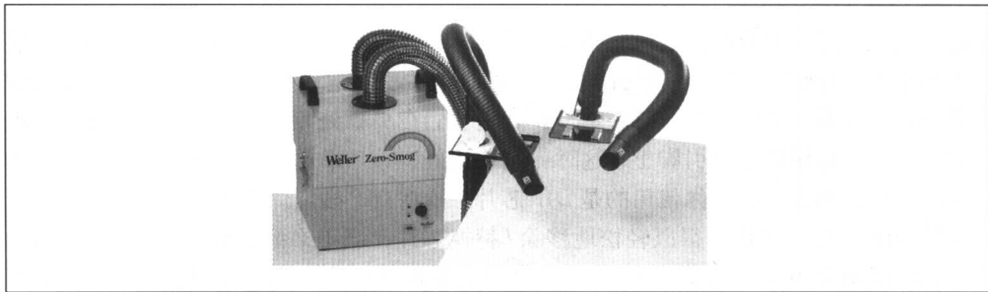


图6-10：Weller排烟机

如何做好焊接

做好焊接的关键是控制热量和流到元件引脚上的焊锡量。太高的热量会损坏元件（特别是灵敏的集成电路），并可能使焊锡过热。请参看技术手册来确定元件能承受的最高温度（和持续时间），并确保你的烙铁温度不会超过它。温度可变的烙铁允许自己设定温度，因此可以避免过热。烙铁的末端尖锐才能做出好的焊脚，一个末端既大又秃的旧型号的烙铁是不合适焊接电子元件的。

警告： 焊接PCB板时一定要确保电路板没有通电，因为焊头是接地的，让它与带电的焊盘接触是很糟糕的。

同样，当插入或拨出插座上的元件时也要让系统断电，大多数的半导体元件不支持热插拔。

应该有足够的焊锡让元件与焊盘间接触良好，但是不能太多以至于焊锡凸起影响美观，甚至于接触到旁边的引脚引起短路就更糟了，如图 6-11 所示。

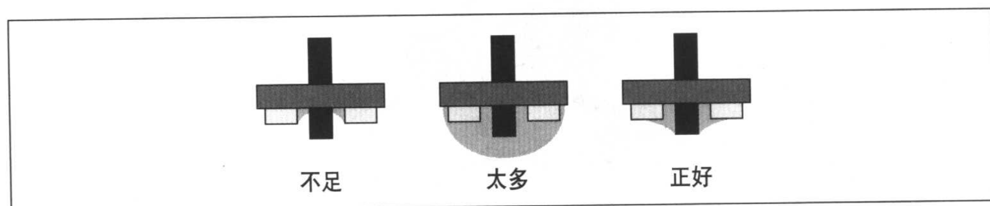


图 6-11：焊于 PCB 上的元件引脚

注意： 在阿波罗 / 农神（Apollo/Saturn）火箭发射期间，美国国家航空和航天局（NASA）发现，焊接方法正确可以减轻好几百磅的起飞重量。

在焊接过孔元件（如DIP封装芯片或连接器）时，要把元件放进孔内，并保证正确安装和表面平滑。开始时，只焊一个引脚，在焊接余下的引脚之前检查元件是否还在正确的位置上。一只手拿烙铁，另一只手拿焊丝，让它们在引脚处相碰，从而进行焊接。在很短的时间内，焊锡将流到引脚上，这时就能连接好，一旦焊锡开始流动，就要把烙铁和焊丝从引脚上移开。如果你使用的是DIP芯片，请将DIP插座焊在电路板上，而不要直接焊接DIP芯片。这让你得以轻松地移除 / 替换芯片，而不必经过麻烦的解焊 / 焊接手续。图 6-12 所示为一个插座（左边）和一个双边插座（右边）。尽管圆孔插座比较贵，不过也比较可靠和耐用。

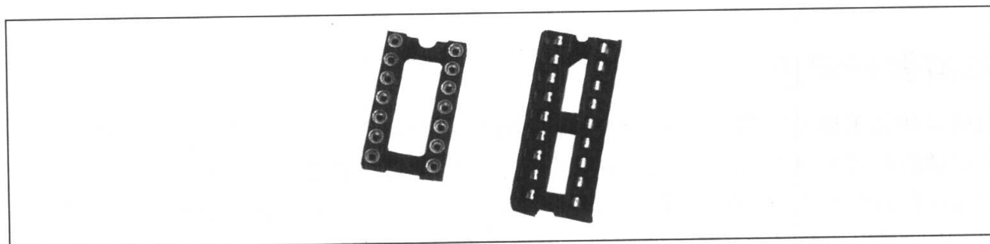


图 6-12：芯片插座

焊接的一般错误是在焊之前把元件加热几秒（使得元件过热），或者直接把焊丝触到烙铁上然后将融化的焊锡涂到引脚上。

焊接贴片元件的程序有所不同。如果你使用的是拆焊台工作，就要用到锡膏。锡膏是装在大容器里出售的。锡膏易干，因此在不使用的时候一定要封好。在焊接表面贴片芯片之前，沿着 PCB 板上的焊盘的每一行挤一条细长的锡膏。太多的锡膏会流到芯片的下面，并且在焊接的时候会松软，因此要轻轻地敷上，以后不够时可以再一点点地加。将芯片置于其 PCB 焊盘上，保证连线正确，然后用拆焊台加热空气（如图 6-13 所示）。太大的空气流会使芯片偏移其正确方向，或者把焊膏吹到下面，由于焊膏是导电的，这对 PCB 板来说很糟糕。太少的热量又会导致焊接连接不紧，然而太多的热量又容易过热并损坏芯片，要做到恰到好处的确是一门艺术，最好在真正开始焊接芯片之前先做大量的练习。

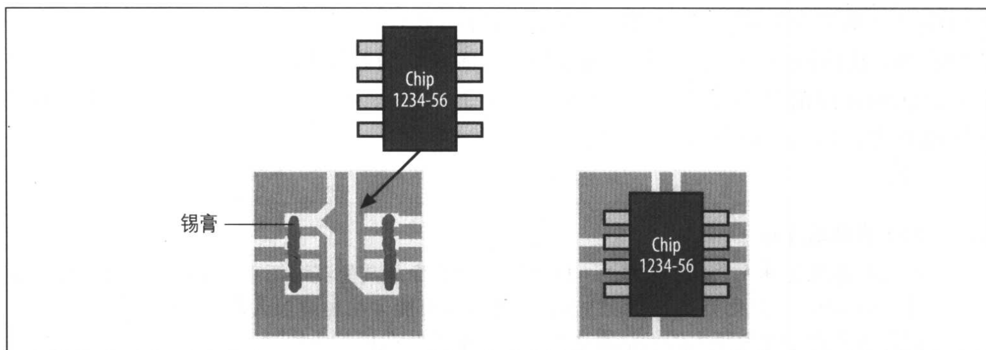


图 6-13：用拆焊台焊接表面贴片元件

表面贴片芯片可以用标准的烙铁焊接，但是对细微空间芯片引脚我们不推荐使用烙铁焊接。与拆焊台技术不同，烙铁焊接员在芯片处于正确的位置上才开始加焊膏。开始时，在将芯片放到 PCB 板上之前，用烙铁和焊丝（或焊膏）直接流少量焊剂到将要安装的元件焊盘上。将芯片放到正确的位置上，仔细矫正，将引脚放在前面所说的少量焊剂上，然后用烙铁加热引脚，焊剂熔化后就将芯片固定在适当的位置上。再次检查芯片的放置，保证它没有产生偏移。如果偏移了，就重新加热引脚，仔细将芯片移到正确的位置上。一旦你对芯片的位置满意了，就可以挤出一小条焊膏到每一排引脚上，并尽可能远离芯片边缘。太多的焊膏会流到引脚之间造成短路，因此要保持薄薄的一层。轻捷地让烙铁的尖端在每一排引脚上走一遍，同时焊膏融化并将芯片固定到 PCB 板上（如图 6-14 所示）。

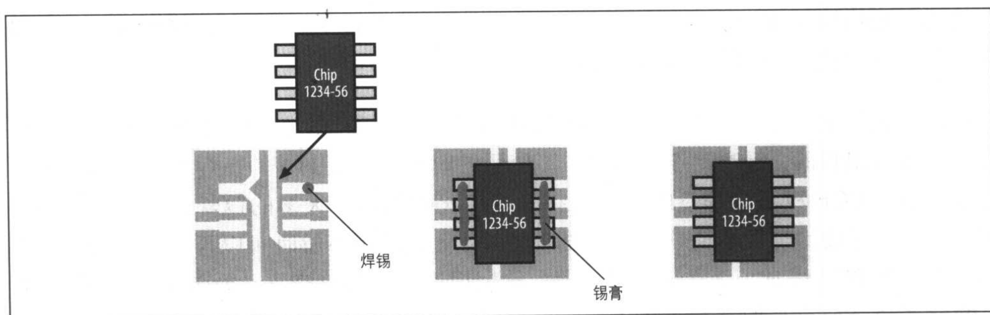


图 6-14：用标准烙铁焊接表面贴片元件

焊锡是一种金属合金，并混合了一定量的助流剂。当加热焊锡时，助流剂就与焊剂分离，流到PCB的边缘，留下一薄层褐色的残渣。残留的助流剂可以用专门的溶剂除去，在大多数的电子供应商店都可以买到这些溶剂。助流剂清洗时会很脏，因此使用时要远离皮肤和衣物，且应该在通风良好的场所使用。助流剂残渣移除仅仅是为了美观，它能使你的产品在顾客面前显得更加专业。但是，这个工作只是为了外观原因，并且溶剂对你和环境都有害，所以应该尽量避免使用。

注意：关于发音的注释

如果你是北美洲人，焊剂读作“sodder”，而如果你是其他英语语系地区的人，这个词应读作“sol-der”。因此美国人应该注意到非美国人可能无法理解什么是“sodder”，相反，他们会认为你是在说陌生的无法表达的事情，而不是在谈关于焊接。

那么，在记住了上述内容之后，你对于构建计算机有什么想法？构建一台计算机（或其他任何电路）的方式有好多种。现在我们来看一下有哪几种方式。

快速的构建方式

仅通过在任意大小的空间里把元件焊接在一起也可以构建非常简单的电路。例如，将晶体放置在处理器上方，晶体的引线直接焊接到处理器的引脚。引线焊接到处理器的引脚上，从而引入接地和电源，把处理器的I/O与外部世界连接起来，如图 6-15 所示。

这是一种非常草率的方法，只对构造极其简单的电路的快速原型有用。我们并不推荐这种做法，但在必要时你可以采取这种方法。对于稍微复杂的或者以某个合理速度运行的所有设计，就不应采用这种方法了。否则，你将花费比调试实际设计或代码多得多的时间去调试硬件构建！

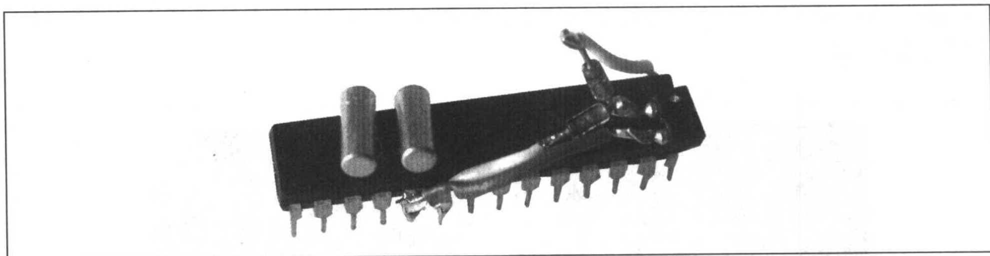


图6-15：一个状似鼠窝的小型微控制系统

面包板

面包板又称电路试验板，是排列着孔的塑料块（如图6-16所示）。它们被设计用来盛放DIP封装的集成电路和离散的元件。术语“面包板”可追溯到很久以前，当时真空管无线电设备是建立在一块结实的木板（用于切面包的木板）上的。这一术语至今仍然采用，在电子爱好者的储藏里，甚至在某些大学的教学实验室里，仍能找到现代面包板。

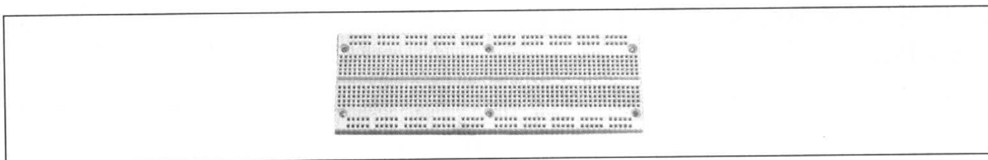


图6-16：面包板

一般而言，使用面包板不是件好事，你应该尽量避免（就把面包板看作COBOL语言在硬件方面的等价物吧）。面包板受制于过多电容、串扰以及噪声敏感度，这使得它完全不适合于构建微处理器系统。长期使用后，面包板也会有机械故障（导致短路）。面包板上的电路连接是通过一小段一小段的电线来实现的，而这些电线便构成了非常小的天线，这些小天线会收集所有零散的电磁辐射，并将其直接传到你的电路里！通常，微处理器并不喜欢Classic Rock FM被调变到它们的数据总线上。

使用面包板绝对不是搭建健壮可靠的系统的方法。如果确实必须那样做，那么你可以在面包板上构建ATtiny15或PIC12C805计算机，但要使用其内部的RC振荡器。对于用到晶体的或带有快速切换的数字信号的任何设计，我都建议避免使用面包板。

绕接

绕接曾经是常用的电路构建技术，现在很少使用了。绕接必须使用DIP封装集成电路，

通过很长的引脚（0.6 英寸）安装在插座上。如图 6-17 所示为一个以绕接方式构建而成的嵌入式计算机。

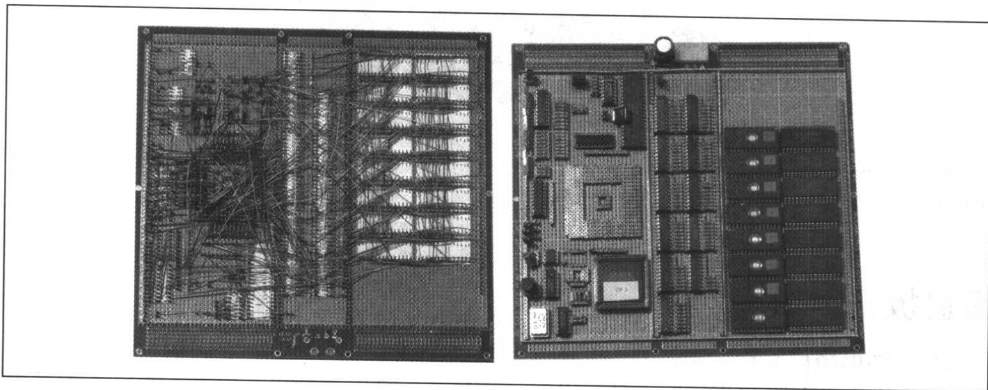


图 6-17：以绕接方式构建的嵌入式计算机的底部和顶部视图

有专门的工具（叫作绕扎工具）可以让你快速有效地在引脚周围绕上细线。图 6-18 所示为一个低价位的手动绕扎工具（左边）以及一个剥线器（右边）。你也可以使用自动的绕扎工具。

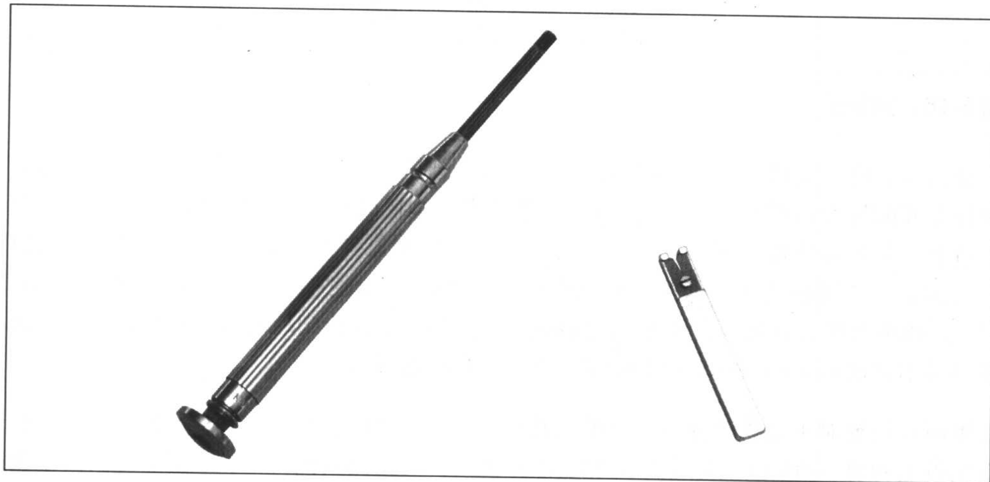


图 6-18：绕扎工具与剥线器

剥线器用于从细线上剥除绝缘外覆，如图 6-19 所示。

线剥好后，接着把它插入绕扎工具的缺口（如图 6-20 所示），然后把该工具放在插座的引脚上。工具旋转，这样线就被绕在引脚上了。

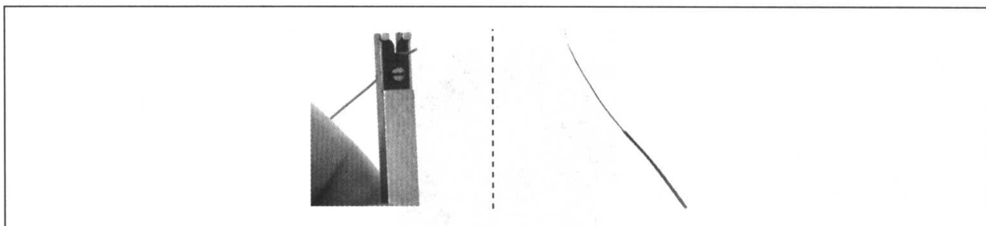


图 6-19：剥线



图 6-20：将线插入绕接工具

DIP 封装的引脚的横截面为正方形，在引脚的周围绕线便构成了冷焊（cold weld），即不带焊接的一种线与引脚间的紧密电气连接（如图 6-21 所示）。这样，通过在系统内对每一点的连接进行绕线，便构建起一个电路。

图 6-22 所示是一台绕线式计算机底部的照片（这是由 Peter Paine 所设计和构建的 LISP 引擎）。注意，引脚的正方形横截面被线紧密环绕。

线接是一种非常快速的原型技术，且非常健壮和可靠。在早期，NASA 惯于使用绕接技术来构建宇宙飞船电子设备，并且许多大型计算机也是使用这一技术来构建的。绕接对于原型设计（尤其是如果你还不清楚设计的最后样子，并且预期会对硬件进行大幅改动）或构建一次性设计，是比较适合的。如果你打算构建不止一台基于自己设计的计算机（你或许会这样），那就跳过绕接好了，直接在 PCB 上搭建它。

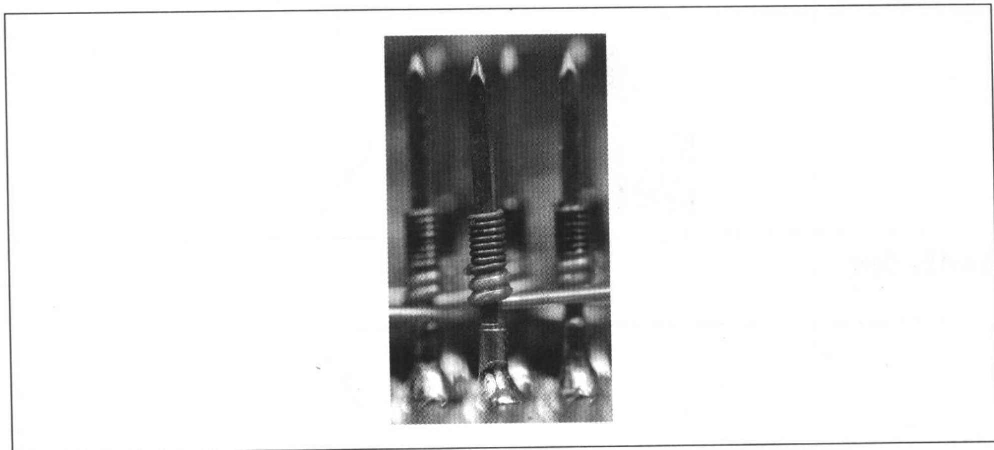


图 6-21：绕线引脚的紧密缠绕（图片由 Peter Paine 提供）

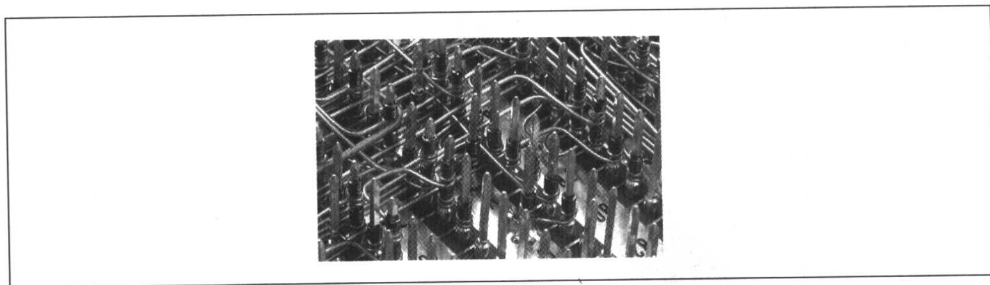


图 6-22：绕线电路的紧密缠绕（图片由 Peter Paine 提供）

印制电路板

印制电路板（printed-circnit board，PCB）是由环氧树脂粘合而成的镀了铜的玻璃纤维板。蚀刻掉部分镀铜，只留下以构成电路互连通路的铜层走线（如图 6-23 所示）。PCB 非常可靠，如果你打算生产不止一个系统，PCB 便是唯一的选择。你可以蚀刻自己的 PCB，但商业 PCB 产品也不是很贵，使用专业生产的电路板所付出的开销是值得的。

可以使用电子设计自动化（Electronic Design Automation，EDA）软件来创建电路原理图和PCB设计。最为流行的EDA软件来自 Mentor Graphics公司（*http://www.mentor.com*）和Protel公司（*http://www.protel.com*）。也有来自 GNU（*http://www.gnu.org*）的 PCB 编辑器（叫作 PCB），它可以免费获取。这些软件通常与几个工具一起发布，提供电路原理图绘制、网表生成（连接与被连接之间的一个列表）、PCB 布局设计、手动布

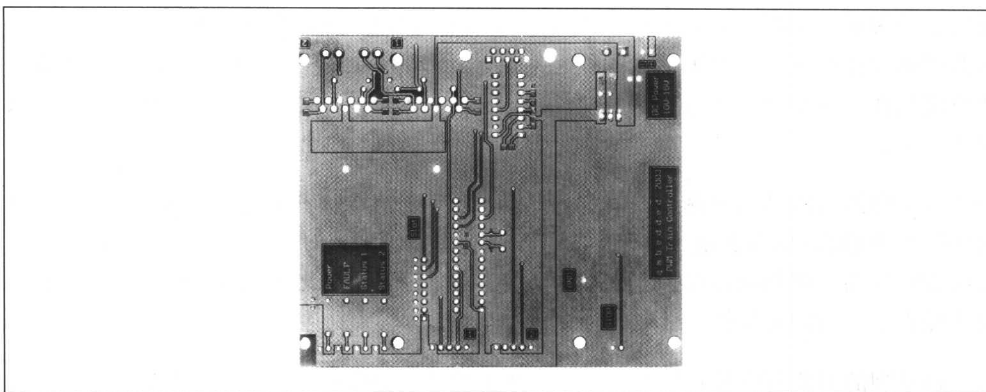


图 6-23：此 PCB 为以 PIC 为基础的模型火车控制器

线（制作连接）和自动布线功能。使用自动布线具有很大的诱惑力，因为它通过让你的工作站去做繁重的布线工作来简化 PCB 的生成过程。然而，我更喜欢亲自布置电路板（我见过一些自动布线软件囫囵吞枣地进行一个设计）。对电路板进行手工布线可能要花费些时间，但这种额外付出是值得的。手动布线能够让人全身心地投入，就像花时间深思一样（那是非常虔诚的）。

PCB 可以是单层（单面）、双层（双面）或者四层、六层、八层、十二层甚至更多。印制板的层数越多，连接的布线就越容易，但随着层数的增多，玻璃纤维的成本也就相应增高。此外，调试一个双层板也远比调试一个十二层板容易。如果PCB有专用供电层和接地层，你的系统就会有更强的噪声抗扰度。专用供电层和接地层对于低端 8 位系统可有可无，但对于高速计算机则是必要的。

多层板是通过通孔来实现的，这意味着板上的小孔有金属连接，从而把不同层上的走线适当地连在一起。阻焊剂是涂在电路板上的（通常是）绿色覆层，以防止在制版过程中焊锡在焊盘和走线之间流淌。定购不带通孔和阻焊剂的PCB是可能的，但你为此所节省下来的金额少得根本不值一提。

印制层（Overlay，也称丝印层）印刷在电路板表面，它标有元件的标号（如R30或RAM4等），以指明在构建系统时元件应该放置的位置。印制层是可选的，如果电路板由其他人来手工焊接元件，那么这一层是很有用的。如果是你自己焊制电路板，那么不用这一层也行，还可节省一些费用。

注意： 避开使用印制层的一个窍门是，把元件信息作为文本放在覆铜层上。但要确保没有与电路走线接触！

电路板的表面铜层叫作*Top*层和*Bottom*层（不足为奇）。传统上讲，由于元件放在顶层而其引脚焊在板下，因而Top层叫作元件层，而Bottom层叫作焊接层。然而大多数现代电路板在这两个层上都布有元件和焊接。这样，术语“元件层”和“焊接层”也就很少使用了。

对于多层电路板内部还有铜层，如Mechanical层（表明特殊机械特性）、Keepout层（给出实际PCB形状）以及其他一些层。在四层板设计中，通常使用外部铜层来作为信号层，而内部层作为电源和接地层。这不仅提供了屏蔽，而且最小化了电流环路面积，从而为设计提供了更高的可靠性。

有五种类型的对象可以放在铜层上，分别为走线、单个焊盘、元件（组合在一起的一批焊盘）、过孔以及覆铜。

走线用来互连元件。走线的宽度用mil或mm表示（1mil=0.0254mm）。走线的厚度可变，通常一块PCB也会有不同宽度的走线。走线越宽，它所能负载的电流就越大。走线越窄，在给定的电路板空间就能容纳更多的走线，从而在PCB上布线就越容易。表6-1给出了不同宽度走线的电流负载能力（1盎斯的铜），规定温度上升为摄氏+10度。

表6-1：走线宽度与电流

Mil	mm	Amps
8	0.2	0.5
12	0.3	0.75
20	1.25	2.5
50	1.25	2.5
100	2.5	4
200	5	7
325	8.12	10

在制作PCB时要看看PCB制作厂商能够加工的公差。如果当地PCB厂商只能生产走线宽度为8mil的电路板，那么你要他们制作一块走线宽度为4mil的电路板就没有什么意义了。

焊盘用来放置元件的引脚，其形状可以是圆形、方形或椭圆形的。焊盘包含一个引脚孔和铜层外围。例如，一个DIP封装元件的焊盘会有多层焊盘，也就是说焊盘在所有铜层上都会出现，因为引脚孔穿过整个PCB电路板。一个贴片封装元件的焊盘只出现在一个层上，如图6-24所示。一列排在一起的焊盘就构成了元件的封装，叫作footprint。贴片

元件引脚孔的直径为零（换句话说，PCB上根本就不会出现引脚孔）。贴片元件的引脚呈“鸥翅”状，它们是被平放焊于PCB板上的。与旧式的DIP封装相比，贴片元件不易受电磁干扰的影响。然而，DIP（双列直插封装）元件很容易安装在插座上，因而在PCB调试期间元件也能很容易移去。DIP封装的元件在早期开发期间有时更具优势（尽管不总是可行），而在批量生产的时候只能选用表面贴片元件。贴片元件辐射的噪音较少，也不易受外部干扰的影响。与之相反，DIP封装的芯片的噪声相关特征很差。

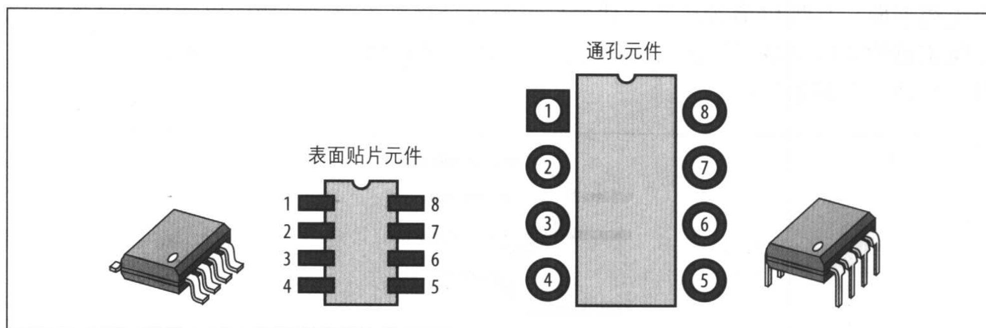


图6-24：表面贴片元件和通孔（多层）元件的footprint

接入焊盘的走线应该直接走向焊盘的中央，如图6-25所示，而非如图6-26所示那样。

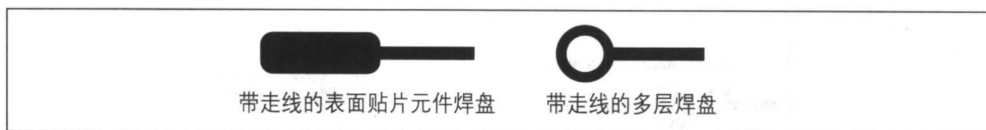


图6-25：表面贴片元件和通孔（多层）元件的焊盘



图6-26：表面贴片元件不正确的走线方式

当为一个元件指定焊盘时，一定要确保焊盘的尺寸足够大，以便有足够的焊接空间来容纳元件的引脚，也要确保通孔的大小（对于通孔元件）可以放得下引脚。一个标准的DIP引脚会轻松地插入一个0.7mm的通孔，而一个DB连接器的信号引脚则需要0.9mm的通孔，其安装引脚需要3mm的通孔。

警告：不要假设你的PCB CAD安装包里的库提供的焊盘、间距或通孔的数据是正确的。安装包所提供的值与实际值大相径庭也不足为奇（不是开玩笑）。再也没有比你拿到一块美观的、崭新的PCB电路板却发现根本放不上任何元件更糟糕的了！因此，对于焊盘、通孔、线宽的尺寸要检查、检查再检查。

当在一个焊盘周围布线时，要确保有足够的空隙，如图6-27所示。走线通常以45度改变走线方向。有些PCB编辑器软件允许进行设计规则检查（也叫作电气规则检查），以确保正确的空隙和没有潜在的短路存在（不能保证这样做了以后电路板就不会有问题，但这却是一个开始！）。

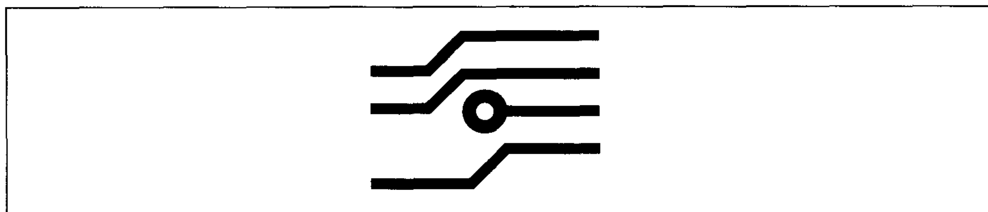


图6-27：在元件焊盘周围走线

避免走线的直角转弯和布线过密，如图6-28所示。

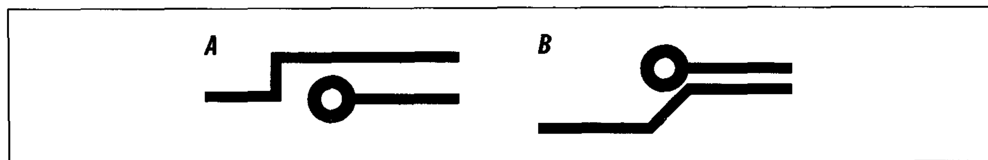


图6-28：如何避免在焊盘周围不正确地走线

表面贴片元件过于紧密的焊盘会产生问题。从表面贴片元件引脚引出的走线如果过于紧密，就可能不起任何作用了。解决这一问题的办法是散开引脚的走线，从而为走线提供更大的空间。图6-29是这种方式的简单示意图。

过孔用于将不同层上的走线连接起来，如图6-30所示。实际上，它们是一些小焊盘。过孔有两种方式：其一是穿越所有层的通孔方式，如图6-31所示；其二是盲孔方式，它只穿过它所连接的层和经过的中间层。使过孔尽可能地小有助于PCB布线，但你应该看看PCB生产商到底可以做到多小。切记确保过孔的外径比其孔径足够地大些，这样在制板过程中就不至于把整个过孔给钻掉。如果空间允许，一个窍门是保持过孔的孔径为0.4mm。这样的话，如果PCB的布线里有bug或制作失败了，就可以使用过孔来焊入绕接线，手工建立连接（或重新搭建连接）。

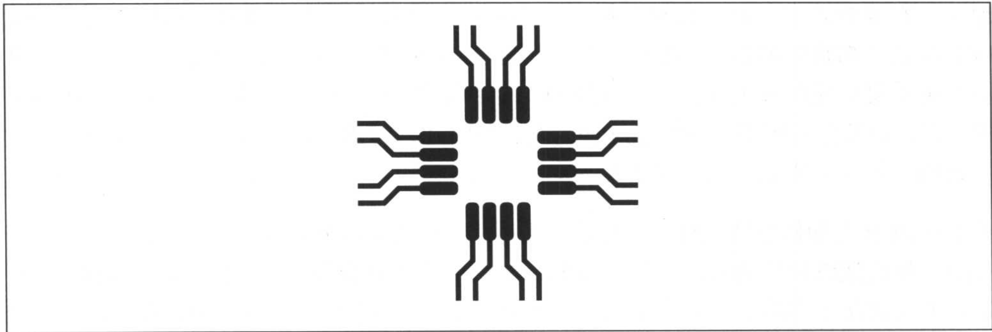


图 6-29：从表面贴片元件上散开引脚

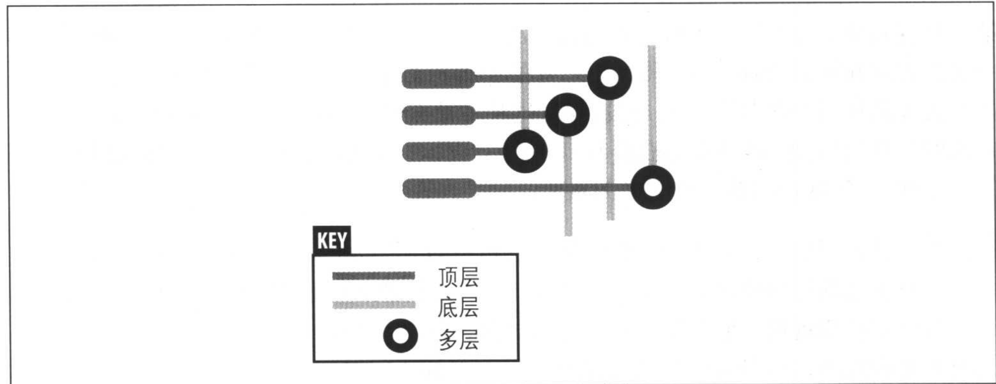


图 6-30：过孔连接 Top 层和 Bottom 层走线

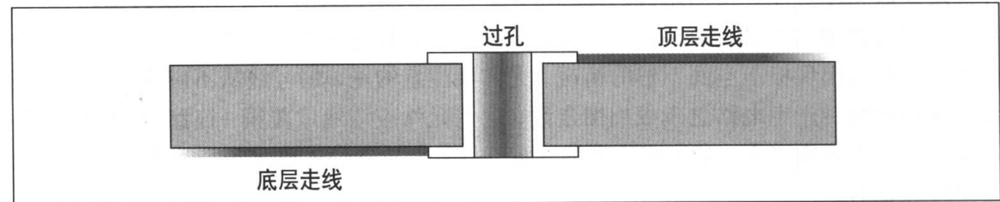


图 6-31：过孔横截面

覆铜用于为PCB某些部分提供屏蔽，也可以用作大量电流的电路通路。接地覆铜通常置于电路的模拟部分的周围，以使其免受数字电路串扰。

为 PCB 布线

首先应该指出的是，当为一个 PCB 布线时，也即有人（或自动仪）将要对 PCB 进行装

配了。在受到将所有东西填充到尽可能小的空间这样的诱惑的同时，请记住用来构建PCB的元件都是受限制的。这不是说你应该尽可能地把PCB做大，而是说大小要符合现实。也不要把元件和走线恰好放满到PCB边缘，要在PCB周边预留出5mm（200mil）的空间。如果你的PCB要装到盒子里，或者要以某种方式安装起来，那一定要确保尺寸的正确，此外不要忘了添加安装孔。

关于在PCB上如何放置元件（尤其是集成电路）有两种类型的观点。第一种观点是，所有的元件应该以同样的方向放置。例如，每一个芯片的引脚1应该指向PCB的左上角。这种方式简化了电路板上元件的装配（或填充），尤其适合于由合同制造商在自动生产线上装配电路板的情况。如果你不是手工装配电路板，元件方向不定的PCB可能会增加生产开支。有些人认为采用这种方法还可以使电路板看起来整洁。

第二种观点是，为了优化布线过程而自己决定芯片的放置方向。不同芯片的插脚引线不一定都要按相同的方向布置，将一个芯片的引脚相对于其相邻芯片旋转90度或180度可能会大大简化两个芯片之间的走线布线。这种方法会使电路板尺寸较小，过孔数目较少，走线的长度也较短，进而能够获得较低的PCB成本、较低的噪声、较低的串扰以及较好的抗扰度 —— 这在高速系统里尤为重要。

无论你打算怎样选择元件方向，都要把一组相关的元件放在一起。把电压稳压器及其支持元件放在电源连接器附近，而任何模拟电路（如传感器或放大电路）都应该尽可能放得离电源连接器远些。把芯片按功能分组放置，布起线来就会容易得多。这看起来像是显而易见的道理，但令人惊讶的是却常常被人忽视。

在进行PCB设计时，应该首先对时钟和高速信号进行布线，以确保它们的走线从起点到终点端采取了尽可能直接的路线。应当在电路合适的地方放置屏蔽（覆铜），以将这些高速信号与PCB的其他部分隔离开。屏蔽应该在布置其他线路之前布置，之后再对其进行布置可能会出现板上空间不够的情况。尤其要注意的是，信号线从不应该在晶体、振荡器或任何时钟产生电路之下或周围走线，这些元件应该通过覆铜（连到接地）从电路的其他元件中隔离出来。晶体应该平放（而不是竖直放置）在PCB上，其下应放置接地面以避开辐射。

对于高速信号，一定要确保紧挨着走线有一条接地回路，这样可以使电流环路面积最小。在高速走线之间预留尽可能多的空间。让两个高速转变的信号线紧密靠在一起会导致串扰，从而引起不可靠的操作。每一走线都会有一个固有的阻抗（电阻）；虽然这个阻抗很小，但它也会影响到快速信号的传输。尤其是，过孔或锐角转角表示沿着走线有一个阻抗改变，而阻抗的改变会导致信号反射。因此，保持过孔数量尽可能少以及在走线中避免直角转弯非常重要。如果一定需要使走线转弯90度，那就使用两个连续的45度转弯来替代。

在高速系统里，你需要连续的电源面和接地面。换句话说，你需要覆盖整个PCB而没有间断的覆铜面。电源面或接地面的任何间断都会导致电流环路面积变大，而这会增加感应和辐射。这就意味着对于高速系统，你的确需要使用四层或更多层的PCB。对于低速微控制器，并非一定要求单独的电源面和接地面，通过在信号层上的元件之间和周围提供覆铜也行。

对总线（如数据总线、地址总线）进行布线时，尽可能地保持信号线平行走线，如图6-32所示。平行走线对时钟信号来说是不行的，因为这可能在相邻的走线里引起串扰，但对总线来说则是合适的。其原因是，总线的信号是一起改变的，而且信号状态会一直保持直到下一事务。接受总线信号的设备只会在总线信号稳定（不改变）时才对其进行采样。因为串扰只会在信号状态改变时产生，因此平行走线的总线毫无问题。通过总线的平行走线，对于每一走线来说信号传播的距离都几乎相同。以完全不同的方式布线的走线会使得线路的长度不匹配，而这就会增加信号时滞，由此导致信号传播时间的漂移。在高速系统里，这种情况会严重地影响信号质量。

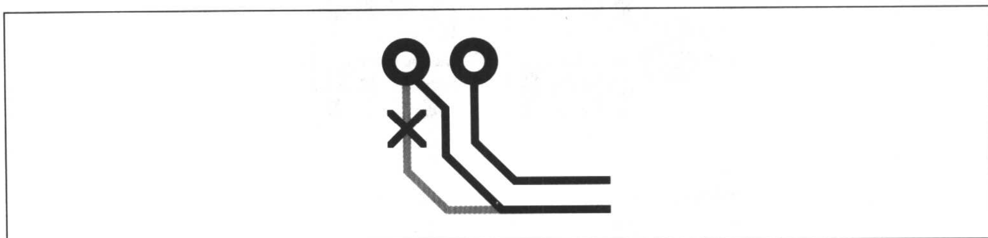


图6-32：保持总线平行以最小化时滞

PCB的短线（stub）是用于连接元件的主走线之后留下的残余线路，如图6-33所示。短线就意味着阻抗不匹配，会导致信号的反射。一个较好的办法是把元件连在主走线上，从而焊盘也位于主信号走线，如图6-34所示。

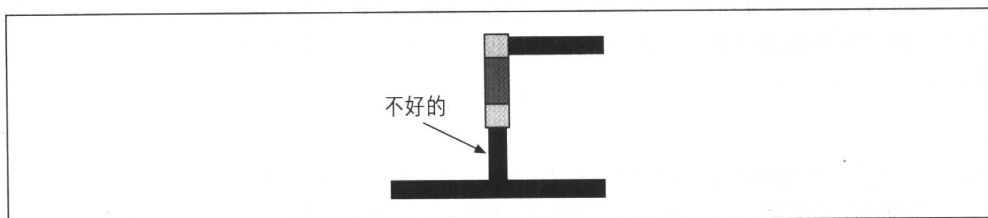


图6-33：避免短线走线

所有电源和接地走线应该尽可能地宽，如果切实可行就采用分离的电源和接地面（层）。如果两者之一出现了，电源和接地（接地终止于电源）应该从信号地或数字地（接地到

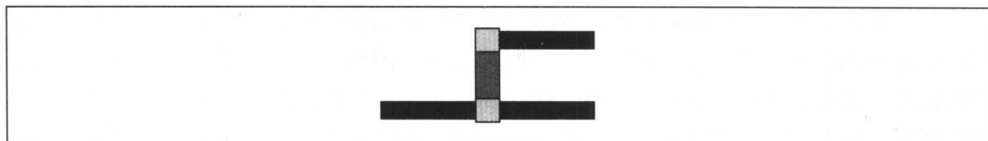


图6-34：如果可能，直接把贴片元件放在走线上

达所有芯片）中分离出来，而电源和接地都得从模拟接地中隔离出来。上述三者都应连在一起，但只能连到一个点。这样做有助于把数字和模拟部分从各自的噪声以及电源噪声中隔离。退耦电容应该尽可能地靠近集成电路的每一电源引脚。图6-35所示为一个双层PCB上的两个元件，其电源和接地走线水平地穿越中间层，其退耦电容紧靠电源引脚。

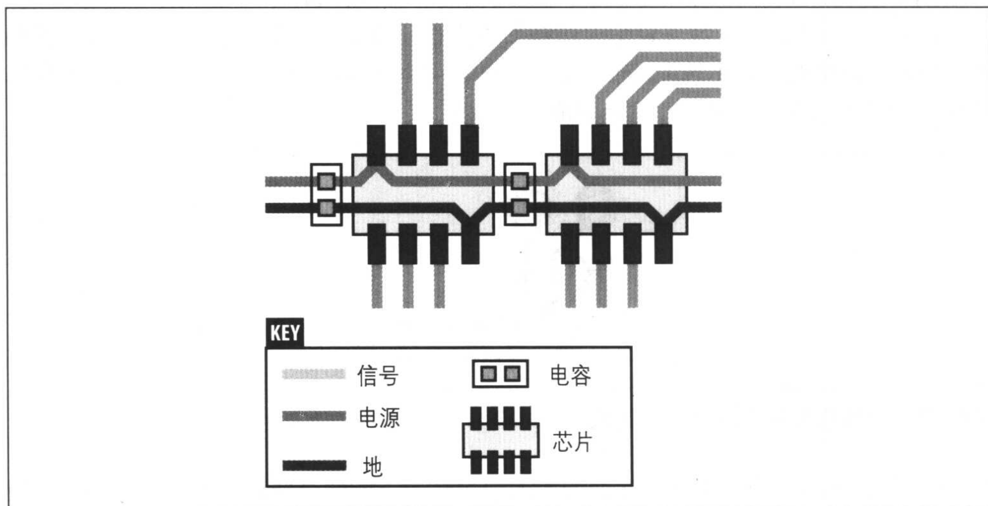


图6-35：退耦电容应该尽可能地靠近芯片的电源和接地引脚

把上面的内容铭记在心后，我们来制作一个简单的AVR计算机电路板。这个设计的细节将在第15章里具体地讲述。在这里，我们仅仅用这个例子来说明制造一个印制电路板的过程。

注意： 一个简单实用的想法是，在接地面（或覆铜）的中央放置单个焊盘，让它从接地面通到电路板面，焊盘呈柱状，用以连接示波器探棒的接地引脚。这样连接会使电路调试过程简单便利得多。你必须使示波器探棒的接地端接地，否则，你不可能得到信号的时序和电压的精确画面（电压是两点之间的势差，因此必须有个参考），这点很重要。

让我们先来看电路原理图，如图 6-36 所示，这个设计将稳压电路、带有一个状态 LED 的 AVR 处理器以及板内编程接口组合在一起。

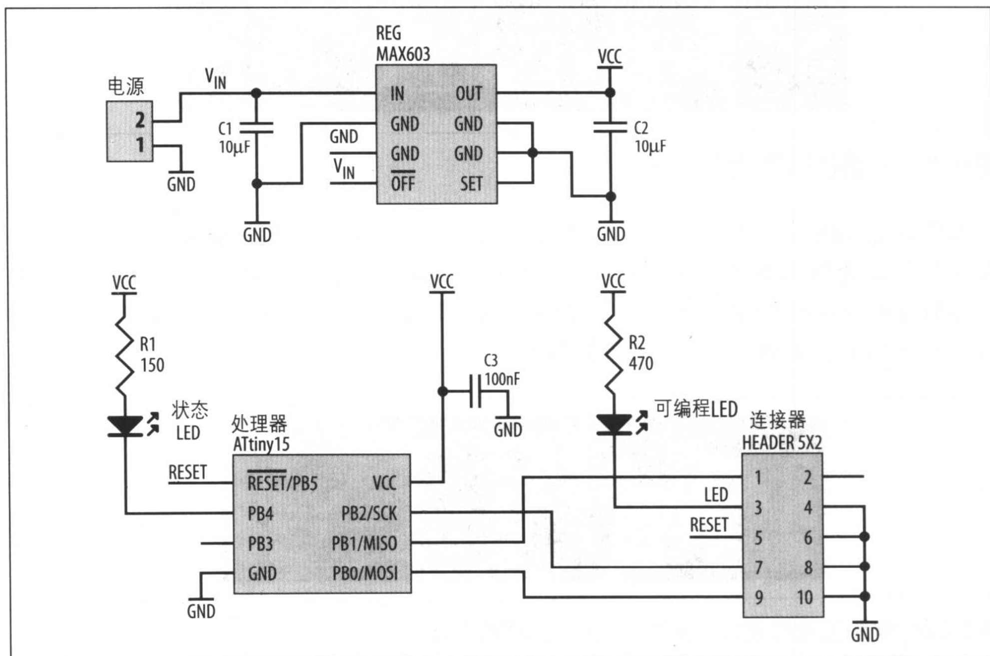


图 6-36：包括 I/O 和编程连接器的完整电路原理图

注意，在连接器的引脚 2 上没有连接，这是由程序员提供的+5V 电压。由于我们的嵌入式系统自身提供+5V 电压，因此就不再需要外部的电源了（如果我们为这个计算机构建的是一个 3V 电压的机型，那么就必须使用程序员的 +5V 电源，同时，我们不得不在编程的过程中使稳压器的输出无效）。

在这个设计中，我们使用电路原理图编辑器产生一个网路表（net list）文件，这个网路表文件告知 PCB 编辑软件需要做哪些互连。

输入到PCB编辑器的网路表文件将自动加载元件引脚信息，这些信息自动重组从而提供最优布置，确保在组件之间的走线最短（如图 6-37 所示）。注意相关元件是如何连接在一起的，例如稳压器、C1、C2 和电源连接器的连接。印制层（OverLay）显示了元件的轮廓。在这个例子中，电阻、电容和LED都是表面贴片封装元件，而两个集成电路则是 DIP 封装。注意有三个焊盘的三角形LED，只有其中的两个连接到了内部LED上，第三个没有使用。这个小巧的电路板的尺寸只有 2" × 0.6"。

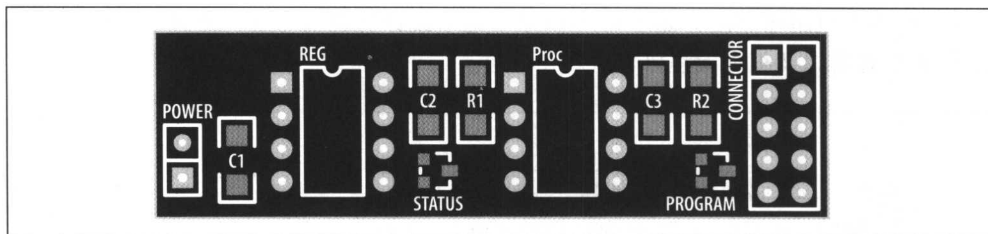


图 6-37：安置在 PCB 上的元件

如果你对元件的布置满意了（这可能还需要进一步调整），下一步就是元件连接的布线。图 6-38 所示为 PCB 的手动布线连接。在本例中，为清楚起见印制层已“关闭”。注意电源与地连接的覆铜（Fill）的使用。对于这个简单的电路，以低速运转且没有外部系统总线，PCB 的布置相对来说也就无关紧要了。

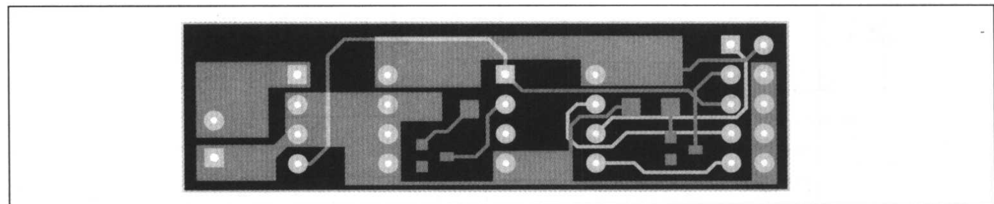


图 6-38：电源和接地连接使用覆铜的手工布线电路板

为了作对比，图 6-39 所示为相同的电路板，但这次是采用了自动布线器来进行连接。请注意，靠近处理器引脚 8 的特殊走线环、奇怪的蜿蜒走线路径以及 PCB 中间不必要的过孔。

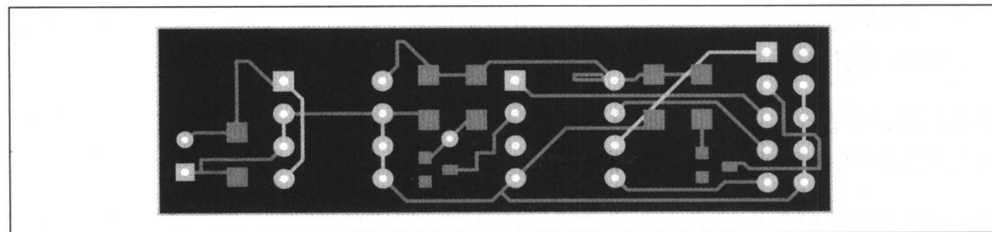


图 6-39：使用自动布线器产生的凌乱结果

这是一个简单的电路板，因此自动布线可以进行大多数的连接。但对复杂的电路板，一般的自动布线器在搞得布线一团糟后，就半途而废了。当然，也有自动布线工具可以做得比这好得多，但它们的价格非常昂贵。

为了增大抗扰度，把一个多边形面放置到手工布线PCB的bottom层作为法拉第屏蔽（Faraday Shield），如图6-40所示。注意多边形是如何围绕元件的引脚“流动”，最后才连接到接地引脚的。在这种方式里，多边形是一个接地面，为系统提供一定（较小）程度的抗扰度。在电路板右边的中间有一块“空白”区，这是由于预布线时的走线使多边形覆铜无法到达该区域造成的。如果设计了一个四层电路板，多边形覆铜将放置在一个单独的层上，覆铜就不会出现不连续现象，除非它途经通孔元件焊盘。

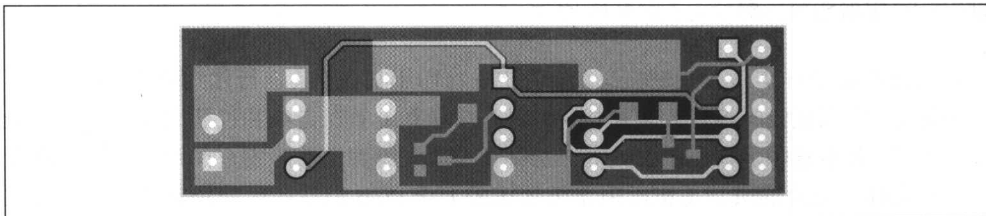


图6-40：带有法拉第屏蔽的手工布线PCB

尽量使用贴片元件（在可能的地方）会使电路板设计做得更小。图6-41给出了相同的设计（手工布线），但是这时是使用贴片元件形式的处理器、稳压器和甚至于更小的电阻和电容。新电路板的尺寸大小只有1"×0.5"。稳压器的焊盘被覆铜掩盖了，因此看不出来（一旦PCB构成，焊盘就会清楚地展露出来！）。唯一的通孔元件是电源连接器和I/O连接器。注意，为了布置走线，电路板需要四个过孔。在以前的设计中，DIP元件的焊盘可以很有效地用作过孔。在均采用表面贴片元件的新电路板里，由于几乎所有元件都在同一层上，就需要过孔来使走线到PCB的bottom层。

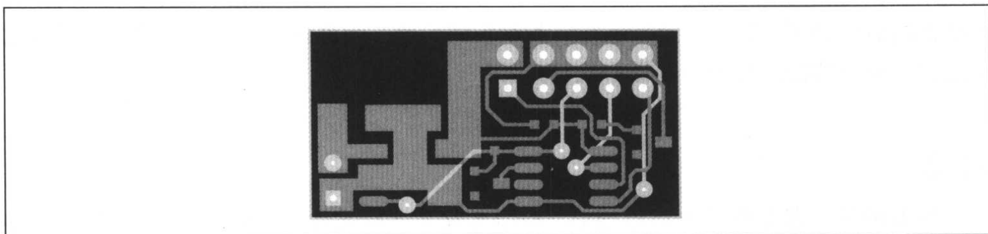


图6-41：表面贴片封装PCB

你可以在PCB两面都使用表面贴片封装元件，以使电路板尺寸更小（可能只要0.5" × 0.5"）。但是，如果你要对其进行专业化的布置，可能会增加构建电路板的成本。

图6-42给出了采用法拉第屏蔽的表面贴片PCB板。

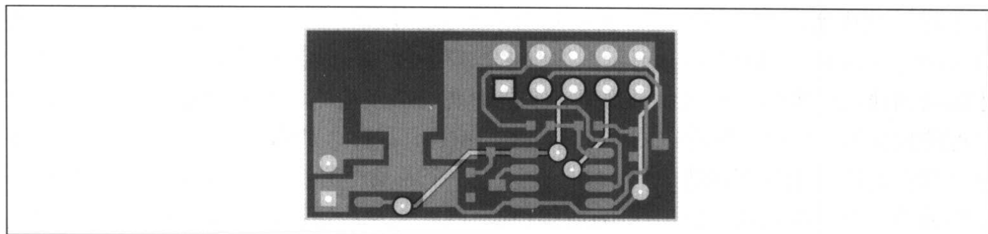


图 6-42：采取法拉第屏蔽的表面贴片 PCB

在将你的PCB设计交付正式制板之前，先把它打印出来，对其进行仔细检查。检查一下安全间距，以保证没有潜在的短路情况。因为从屏幕上看在走线和过孔之间存在的一个微小间隙，并不意味着在制板时此处不会短路。最好提供足够的安全间距以杜绝短路发生的可能性。经验的做法是把间距定为等于或大于 PCB 制造商所能蚀刻的最小走线宽度。你如果硬要把间距弄得更小，就是在自找麻烦。

注意： 当你的PCB设计打印出来后，把实际元件放在图纸上，再检查安全间距。因为仅仅是CAD封装里的元件轮廓不接触，并不能说明实际芯片也不会接触。这个问题在PCB制作前解决比在制板之后解决容易得多。

制作电路板

一旦PCB设计构建完毕并进行了仔细的检查，确保了所有的焊盘和走线都是完整的，而且已正确地蚀刻了，那么此后每进行一步制板的构建，就再做一次仔细的检查。对接地焊盘和电源连接器的接地引脚做一次连续测试。

上电

构建PCB的第一步是焊接电源连接器、稳压器及其支撑元件，也包括“电源”LED（如果你的设计中有的话）。

一旦你已经将元件焊接好，就需要提供电源给电路板上电并检查它的可操作性，同时检查电路板上准备通电的每一个焊盘是否都已经通电，并检查接地焊盘，确保不需要电的地方没有电。

接着，为IC焊接电源耦合电容，再添加处理器的振荡器和耦合电容。如果振荡器是一个模块，就用示波器检测它的输出引脚是否有输出信号。

如果电路板使用了IC槽，就将这些IC槽焊接上去，然后嵌入元件。如果你使用需要外部重编程的处理器，那么用IC槽就很方便。

添加处理器

检查处理器引脚是否有合适的电源和接地连接，然后将处理器焊接到板上（如果用了插槽，就将它插到插槽上），记得在此之前保证断电。如果你的处理器需要在外面调试程序代码，那就在接入电路前调试。上电后再用示波器检测处理器的时钟，确定它在振荡，你应该能看到一个适当振幅的完整正弦波。再对比技术手册中的值测试电压，如果振荡器的振幅不正确（可能是接触不良或局部短路），它就无法驱动处理器。

如果你在构建的系统用了没有外部ROM的微控制器（如本章所举的例子），你首先要编一个测试软件，让处理器的I/O线来回摆动，再用示波器来观察I/O线，判断系统是否在正确地执行代码。如果在你的设计里用了状态LED，就打开它。当看到自己第一次设计和制作的机器的LED闪亮时，你一定很高兴。

一旦确定处理器在软件控制下能工作，就可以开始添加其他的硬件和应用的软件组件。

警告： 在制作过程的任何阶段都不要太冒进。如果你每次只作一个改变或添加一个东西，那么当机器突然不能工作的时候找到其原因就会容易得多。做事情就要一步一个脚印！

关于调试的一些思考

调试如同科学，也是一门艺术。你可以用各种各样昂贵的测试设备去安装一个平台到转换点上，但是如果没有逻辑方法和有条理的头脑，你就发现不了一些难以捉摸的错误。相反，通过“正确的思维”，即使用最少的工具也能把最奇特的bug找出来。但是要测试的系统越复杂，通过调试明确一个错误就越困难，而你所能支配的最先进最有用的调试工具就是自己的大脑。因此，学习调试从某种意义上说就是学习如何思考得更仔细、更有条理。

调试硬件比调试软件更灵活。对于代码，你总能采用某种诊断工具监测程序的执行，但这并不是说软件调试就是微不足道的 —— 这样认为可就大错特错了。而对于硬件，调试就意味着或者是都有用或者都无用。软件的优势在于它能够慢慢地投入执行，而硬件从一开始就有非常多的工作要做。

调试从本质来说就是要确定哪些工作、哪些不工作。由于设计的复杂性越来越高，从而发现硬件和设计中错误的难度也就越来越大了。

举例来说，你设计的嵌入式系统不能从串口输出字符了，有什么原因会造成这种情况呢？可能是程序里的bug，也可能是电缆故障，或者是RS-232C接口芯片坏了，也可能是串行芯片本身都坏了；还有可能就是串行芯片的振荡器的时间问题，或者是电压有问题；也许处理器本身就没有成功重置，也就根本没有执行代码，如果是这样，可能是它的上电重置没有生效或者是电压不足检测器在不该起作用的时候起作用了；或许处理器和芯片之间的一根数据线没有连接；也可能是印刷电路板（PCB）的某个制作问题，也许它的焊接有错误；还有就是可能你的稳压器操作有误，给芯片提供了不合适的电压。这些只是一些首先能想到的很明显的可能原因，但是还有上千个危害甚大的深藏的原因。

一个问题也有可能是多个原因造成的。解决一个错误的最好方法就是将问题隔离，“20种问题法”便是完成此事的最好方法，用分而治之的办法来解决。

还是举上面的串行端口错误的例子。当你第一次试着去测试串行端口的时候你就发现了这个问题，你的简单测试程序不能输出字符，这是软件的原因还是硬件的原因？如果是硬件的原因，那么究竟是电缆的原因，还是串行芯片的原因，或者是核心系统上更为基础的原因？于是测试电缆和终端（或主机）。首先断开电缆和嵌入式系统的连接，然后用一片金属（一个螺丝刀就可以）将电缆连接器上的第二个和第三个引脚（**Rx** 和 **Tx**）短路。现在你再在终端（或主机上的终端程序）上敲入一些字符，通过短路，终端显示屏上应该显示回刚才敲入的内容，这样你就可以确定是不是电缆错误和终端设置不正确的原因。

如果屏幕显示正确，那么问题就出在嵌入式系统了。换一个比较简单的端口测试程序（摆动数字I/O线或让LED闪烁），这个简单的动作就能告诉你很多信息（阿基米德说过“给我一个足够长的杠杆，我能撬起地球”，那么也可以这么说“给我一个状态LED和足够长的时间，我就能调试世界上所有的程序!”）。它将告诉你处理器是否能正确地执行代码，从而表明处理器和ROM（如果是个分开的芯片）是否电源正常并且通信正确，并进一步表明复位电路、节电检测器、振荡器、稳压器、地址译码器和其他的支持逻辑是否无误。如果其中任一项错误，那么处理器就不能执行代码，并因此I/O线不能摆动，LED不能闪烁。用这种简单的测试你就能排除很多可能的错误。

如果这种测试失败了，你就应该进一步找原因，比如检查振荡器、复位电路或电压稳压器等等，各个击破。如果上述测试通过了，那么错误只能在串行芯片本身了。大多数的串行芯片都包括一些能够手工设置的数字I/O（比如**RTS**）。写一个简单的测试代码，来确定是否能和芯片会话。如果测试通过，你可以知道去查字符输出程序或者去查RS-232驱动。如果测试失败，那么问题就在与芯片的会话中，于是就可用一个示波器来检查片选信号或其他进入串行芯片的控制信号，看它们是否有效，是否合理。写一个程序，让它不断地往串行芯片的一个寄存器写入一个字节。虽然这个程序的执行对于串行芯片是

没有意义的，但是你却可以通过连续写入同一个数字来观察总线的活动。你的（示意的）实现这个动作的代码如下：

```
        load    r1,#0x55         ; 加载 %01010101
loop    store   serial_control   ; 写
        jump    loop             ;循环继续
```

你期望看到在片选信号和写允许有效的同时，前述的位组合格式发送到数据总线上（而且很重要的是必须和串行端口的引脚一一对应）。

这些将让你能够发现处理器写入芯片的问题所在。如果能够表明可以正确地写入芯片，那么错误就应该存在于软件或者串行芯片与串口的连接。用这种各个击破的方法，你就可以分离出问题的所在，设计测试来检验系统操作的各个方面。

有时你会碰到一些意想不到的错误。有的东西本应该正常工作却没有正常工作。你所测试的每一项都好像是对的，但是总系统却不能运行，这令人困惑。这在于你犯了一个普通的错误 —— 你作了某种假设。对于某个你并不十分清楚的细节，你不自觉地假设它是正确的，事实上却是错的。这是最难克服的。当你对自己说："这应该是在运行的，但它却没有，这应该无关紧要。"的时候，你要对自己说："还有地方我还没有测试到。"于是去搜寻它，如果搜寻不到，那就是你的努力还不够。

阅读电子说明书！

我曾经设计了一个工作在5V电压下的系统，它工作很正常。于是我就用3.3V下工作的同样的零件开发同样的工作在3.3V电压下的系统。这应该是可以工作的，但是却没有。所有的时钟都是正确的，所有的电压都无误，总之我所测试的每一项都是正常的、无误的，甚至没有一点不正常的迹象。我想不明白，因为好像没有什么再需要测试的了。

我清楚这个设计和代码在5V下工作是正常的，在3.3V下肯定有某些特殊的硬件问题。但会是什么呢？处理器电源是通的，稳压器在工作，振荡器在运行，复位电路也是正常的。它应该可以执行代码，然而却连最简单的测试代码都不能通过。

我想，肯定在某个地方我作了一个不合理的假定。于是我花了将近一周的时间来找这个可疑点，它是那样难以捉摸。从5V系统过渡到3.3V系统，我已经选择了3.3V的处理器，我假定（逻辑上正确，但是错误地）节电检测器（内嵌在处理器中的）是工作在正确的电压下。你希望如此，但是它却是错误的。处理器的生产商在生产3.3V的芯片时改变了该设备的操作电压，但是忘了更改节电检测器的电压。于是，

在3.3V的处理器中，节电检测器由于提供的电压低于4.5V而断开。这样，处理器重置怎么也不能通过，因此也不能执行代码。

我作了一个不正确的假定，就是3.3V处理器里面所有的元件都工作在3.3V。如果要让处理器正确操作，就需要使节电检测器无效（这是可选的），但是这并没有在技术手册中进行详述，只能通过阅读电子说明得到提示。

上述经历得到的启示是：不要作任何假定，测试所有的元件。如果还是不管用，那是你的测试工作还不够仔细。

当设计系统和布线PCB时，记住最后必须调试它。于是，在设计的时候就在心里对它进行调试。包括一个或更多的状态LED，这对于调试嵌入式硬件将有极大的帮助。当然，用一个细致的调试器也可以做很多事情（比如gnu的gdb调试器），但是你还是需要在调试器能够运行前让硬件能够有某种程度的工作，状态LED就可以帮助你做到这一点。

你还将需要用示波器来观察信号，所以在电路板上设计一个接地引脚，这样你就可以接示波器。你还必须使示波器的探棒能到达板上的每个电路单元，以测试它们是否工作。如果不能到达某个单元，你就不能保证在某个特殊的信号下它没有问题。

因此，甚至早在设计期间就要想好如何来测试子系统以及如何来分离问题，并且在你的设计中置入必要的支持。

JTAG

JTAG（Joint Test Action Group，联合测试访问专家组），有时也被称为测试访问端口（Test Access Port，TAP），能够提供对处理器内部的访问，并且由此访问系统中的其余部分。JTAG由IEEE标准1149.la-1993中的“标准测试访问端口和边界扫描体系结构”所定义。商业上应用的测试套件使用的就是由JTAG所提供的内电路（in-circuit）调试功能。也可以使用标准文档所提供的信息，自己手动驱动JTAG。

JTAG端口允许硬件和软件的实时调试。通过使用直接运行在目标系统的代码，可以进行单步或多步调试。可以单个地手动切换处理器的信号线，以便测试计算机的外部子系统（即所谓的边界扫描）。既可以在代码域设置断点，也可以在被访问的地址域（或者设备）设置断点。JTAG端口允许外界对寄存器和内存单元进行检测和修改。在使用JTAG端口之前，需要有与JTAG兼容的支持工具。更多的信息请参考IEEE标准1149.la。

使用 JTAG

要使用JTAG，你所选用的处理器必须对它有明确的支持。换言之，处理器必须内部集成了JTAG接口。JTAG在小型微控制器中极为罕见，几乎不支持。然而，较高级的微控制器以及速度较快的处理器通常都会包含JTAG接口，对于使用这些芯片的系统而言，JTAG接口将是其调试过程中不可或缺的一部分。

JTAG使用边界扫描技术探测微处理器以及外围设备和内存之间的电路连接。它通过确认输出（独立于CPU）并读入输入管脚上连接的外设的响应信息来完成这一功能。这不仅能测试PCB间的互连，而且还有助于设计验证，甚至还能校正时序。JTAG能够独立于CPU进行操作，可以“手动”驱动输出，还能查询处理器的生产厂家、处理器类型以及版本号。当一个电路板处于测试状态时，JTAG还能使输出管脚失效。Freescale Semiconductor（前身为Motorola）使用了一个片上仿真器（On-Chip Emulation，OnCE）向JTAG接口添加功能。片上仿真器能够使处理器运行并观察系统对于所执行软件的状态。它还能读取或设置寄存器或存储器中的参数值，并提供多种调试手段（如设置断点、单步执行以及指令跟踪等）。

JTAG端口还提供对状态机的访问，这种状态机能够完成边界测试的功能。状态机有四个寄存器，分别为指令寄存器、边界扫描寄存器、设备识别寄存器和旁路寄存器。

JTAG端口包括四个专用信号（见表6-2）。如果你觉得这些信号看起来好像是同步串行接口信号，那么你就对了，因为JTAG确实是同步串行接口。

表6-2：JTAG信号

信号名	功能
TDI	测试数据输入
TDO	测试数据输出
TMS	测试模式选择
TCK	测试时钟

由于JTAG可以为系统设计者带来许多好处，所以其应用越来越普及。大多数新出品的处理器都在其设计中包括了JTAG。特定处理器的JTAG的使用信息，请参阅制造商的技术资料。

第7章

用SPI添加外部设备

三十辐，共一毂，当其无，有车之用。
埏埴以为器，当其无，有器之用。
凿户牖以为室，当其无，有室之用。
故有之以为利，无之以为用。
——老子
《道德经》

在本章及下一章中，我们将介绍两种简单的接口，用来将外围设备（简称外设）连接到一个嵌入式系统的微控制器上。通过这两种接口，可以连接实时时钟，用于保存参数的非缺失性存储器、传感器接口等等。这两种接口不仅容易使用，而且造价低，这些优点使得它们成为小型嵌入式应用中的理想选择。有些微控制器集成了这两种接口，有些里面则只含有其中一种。具体使用哪一种接口，完全取决于处理器要提供哪些接口和你所使用的外设的要求。

串行外围设备接口

串行外围设备接口（Serial Peripheral Interface，SPI）是由Motorola公司开发的，用来在微控制器和外围设备芯片之间提供一个低成本、易使用的接口（SPI有时候也被称为4线接口）。这种接口可以用来连接存储器（存储数据）、AD转换器、DA转换器、实时时钟日历、LCD驱动器、传感器、音频芯片，甚至其他处理器。支持SPI的元件很多，并且还一直在增加。

与标准的串行端口不同，SPI是一个同步协议接口，所有的传输都参照一个共同的时钟，这个同步时钟信号由主机（处理器）产生。接收数据的外设（从设备）使用时钟对串行比特流的接收进行同步化。可能会有许多芯片连到主机的同一个SPI接口上，这时主机

通过触发从设备的芯片的片选输入引脚来选择接收数据的从设备，没有被选中的外设将不会参与 SPI 传输。

SPI 主要使用 4 个信号：主机输出 / 从机输入（**MOSI**）、主机输入 / 从机输出（**MISO**）、串行时钟（**SCLK** 或 **SCK**）和外设片选（**$\overline{\text{CS}}$**）。有些处理器有 SPI 接口专用的片选，称为从机选择（**$\overline{\text{SS}}$**）。

MOSI 信号由主机产生，从机接收。在有些芯片上，**MOSI** 只被简单地标为串行输入（**SI**），或者串行数据输入（**SDI**）。**MISO** 信号由从机产生，不过还是在主机的控制下产生的。在一些芯片上，**MISO** 有时被称为串行输出（**SO**），或者串行数据输出（**SDO**）。外设片选信号通常只是由主机的备用I/O引脚产生。图 7-1 所示是微处理器通过 SPI 和外设进行连接的示意图。

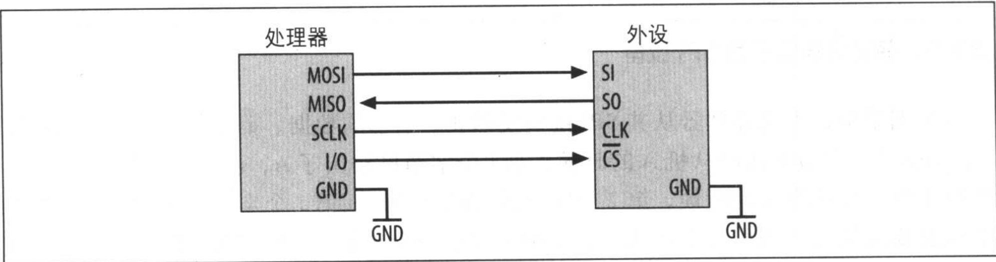


图 7-1：基本 SPI 接口

主机和从机都包含一个串行移位寄存器，主机通过向它的 SPI 串行寄存器写入一个字节来发起一次传输。寄存器通过 **MOSI** 信号线将字节传送给从机，从机也将自己的移位寄存器中的内容通过 **MISO** 信号线返回给主机（如图 7-2 所示）。这样，两个移位寄存器中的内容就被交换了。从机的写操作和读操作是同步完成的，因此 SPI 成为一个很有效的协议。

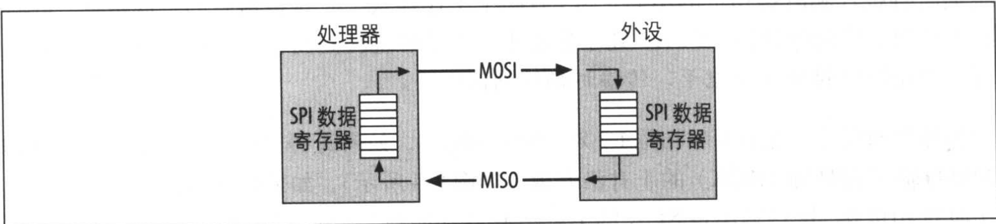


图 7-2：SPI 传输

如果只是进行写操作，主机只需忽略收到的字节；反过来，如果主机要读取从机的一个字节，就必须发送一个空字节来引发从机的传输。

当主机发送一个连续的数据流时，有些外设能够进行多字节传输，许多拥有SPI接口的存储器芯片都以这种方式工作。在这种传输方式下，SPI从机的片选端必须在整个传输过程中保持低电平。比如，存储器芯片会希望在一个“写”命令之后紧接着收到四个地址字节（起始地址），这样后面接收到的数据就可以存储到该地址。一次传输可能会涉及千字节的移位或更多的信息。

其他从机只需要一个单字节（比如一个发给模/数转换器的命令），有些甚至还支持菊花链连接（如图7-3所示）。

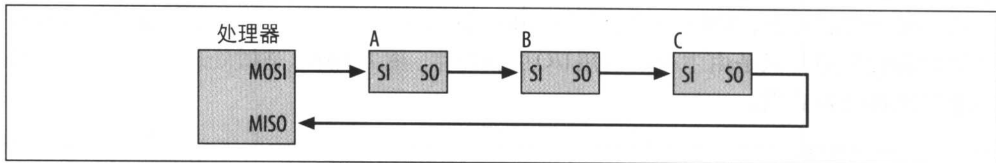


图7-3：菊花链连接三台SPI设备

在这个例子中，主机处理器从其SPI接口发送3个字节的数据。第1个字节发送给从机A，当第2个字节发送给从机A的时候，第1个字节已移出了A，而传送给了B。同样，当第3个字节发给A的时候，第2个字节被传给了B，而第1个字节被传给了C。如果主机想要从从机A读取一个结果，它必须再发送一个3字节（空字节）的序列，这样就可以把A中的数据移到B中，然后再移到C中，最后送回到主机。在这个过程中，主机还依次从B和C接收到字节。

注意，菊花链连接不一定适用于所有的外设，特别是要求多字节传输的（比如存储器芯片）设备。另外，要对从机芯片的技术手册进行仔细检查，确定你能对它做什么而不能做什么。如果芯片的技术手册中没有明确提到菊花链连接，那么这个芯片不支持这种连接的机率为50%。

根据时钟极性和时钟相位的不同，SPI有四个工作模式。时钟极性有高、低两极：时钟低电平时，空闲时时钟（SCK）处于低电平，传输时跳转到高电平；时钟极性为高电平时，空闲时时钟处于高电平，传输时跳转到低电平。

时钟相位有两个：相位0和相位1。对于时钟相位0，如果时钟极性是低电平，**MOSI**和**MISO**输出在时钟（**SCK**）的上升沿有效（如图7-4所示）。如果时钟极性为高电平，对于时钟相位0，这些输出在**SCK**的下降沿有效（如图7-5所示）。**MISO**输出的第X位是一个未定义的附加位，是SPI接口特有的情况。不必担心这个位，因为SPI接口将忽略该位。

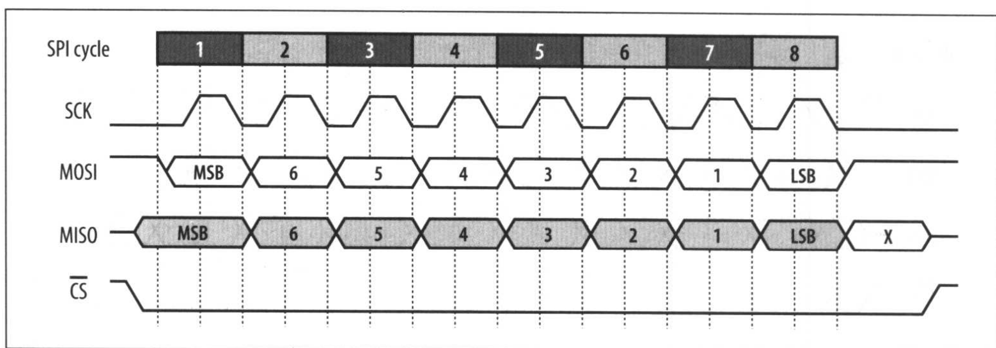


图 7-4：时钟极性为低电平且为时钟相位 0 时的 SPI 时序图

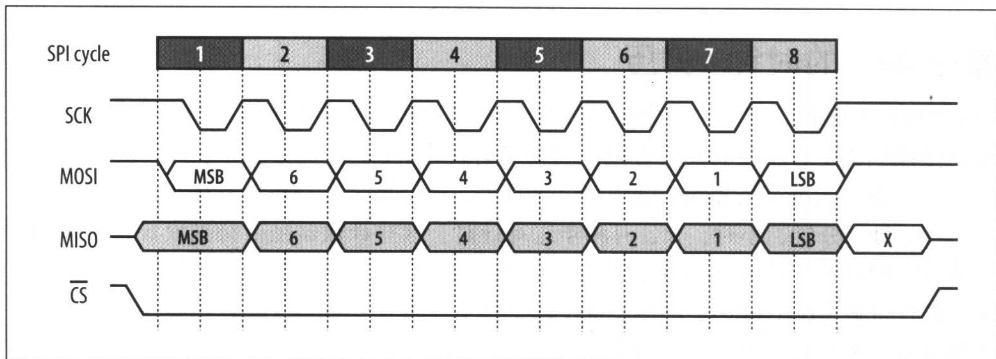


图 7-5：时钟极性为高电平且为时钟相位 0 时的 SPI 时序图

对于时钟相位 1，情况则相反。此时如果时钟极性是低电平，MOSI 和 MISO 输出在时钟（SCK）的下降沿有效（如图 7-6 所示）。如果时钟极性是高电平，这些输出在 SCK 的上升沿有效（如图 7-7 所示）。

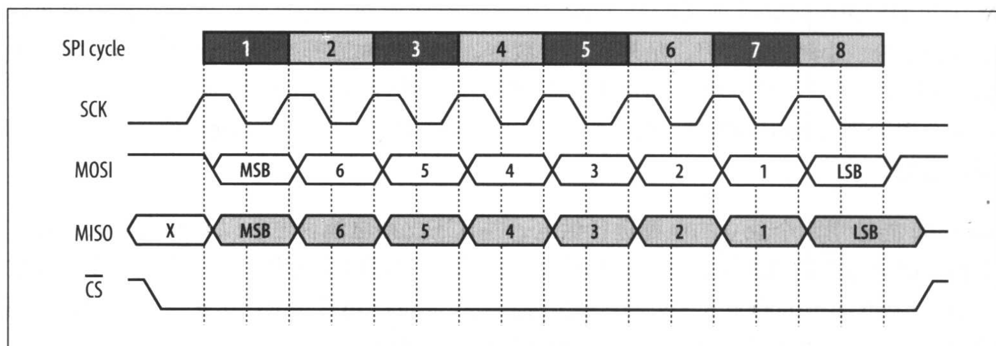


图 7-6：时钟极性为低电平且为时钟相位 1 时的 SPI 时序图

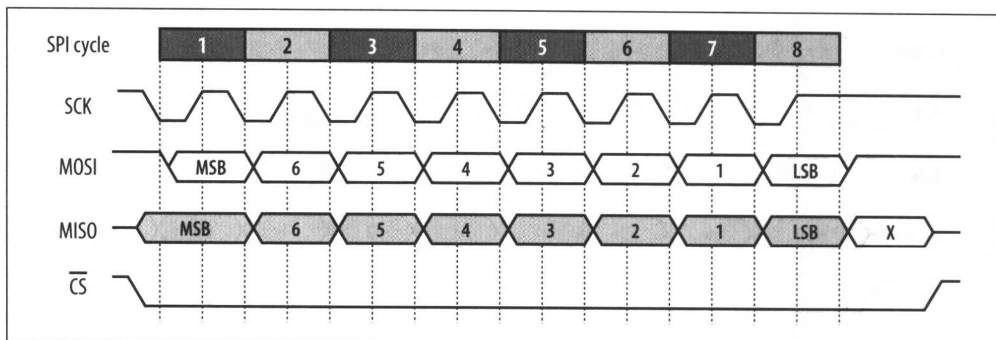


图 7-7：时钟极性为高电平且为时钟相位 1 时的 SPI 时序图

基于 SPI 的时钟 / 日历

SPI 设备种类多样，接下来只介绍其中的几种。同时也看看如何使用 SPI 接口将外设添加到微控制器中，现在先让我们把处理器和时钟 / 日历芯片连接起来。这种芯片包含一个由石英驱动的振荡器模块，就像一个处理器。这个振荡器模块驱动内部计数器，用来对毫秒、秒、分钟、小时、天、月和年进行计时。这些计数器经过特别设计，用来提供精确的时间保持；许多还有附加功能，比如“报警”（处理器会在一个特定的时间被中断）和看门狗（Watchdog）功能。有些还包括电压监控功能，这样时钟芯片就可以充当系统监控器的角色，用以在电源电压波动时通知处理器。时钟芯片的种类很多（并不是都通过 SPI 接口连接），在这个例子中，我们使用的是 Maxim 公司的 DS1305 芯片。

实际上，我们这里所用的连接时钟芯片和处理器的方法与其他 SPI 设备的连接方法是一致的。有些具有 SPI 接口的芯片有特殊的要求，不过大部分都很简单直观，这使得 SPI 成为了一个非常有用的接口，使系统功能的增加变得很容易。

Maxim DS1305 实时时钟（Real-Time Clock，RTC）可以提供时间保持功能，也可以记录秒、分、小时、日、月、星期和年等数据。该芯片知道哪些月份有 30 天，而哪些月份有 31 天，甚至还可以根据 2100 年的情况对闰年进行调整。此外，它可以在一天内产生两个对处理器的中断警报，这些警报可以用来触发一个规定的系统事件，比如备份和用户通知等。

DS1305 能够用两个独立的电源进行供电，并且还支持电池供电，以对其内部状态进行备份。这个芯片的电源电压在 2V~5.5V 之间，使得它可以用多种电源来供电。另外，它还有一个 96 字节的静态 RAM，用来存储参数，你可以用这个 RAM 来保存表示系统模式的变量、安全密码甚至嵌入式系统的认证码，就像台式机的一样。

RAM就如同时间保持功能一样由电池供电，因此只要电池可以工作，RAM的内容就会保存下去。根据所选择的电池不同，这个时间可长达10年。因此，通常情况下，在嵌入式系统的预期寿命内，内部参数都会保存在RAM中。

注意： 如果你开发的是商业嵌入式系统，那么会遇到一些逾期付款的客户，这时可以使用RAM来保存一个授权码。在出售系统的时候，将系统设计为运行45天后关闭，如果客户按期付款了，就可以将正确的授权码提供给他。系统将正确的授权码存储在实时时钟的RAM中，然后就开始正常运行。

DS1305的供电方式多种多样，它有3个电源输入端（**VCC1**、**VCC2**和**VBAT**），通过这些输入端，可以选择电源。**VCC1**是首要的电源输入，直接连到系统电源。当计算机通电运行的时候，DS1305就可以从这里获得电流。**VCC2**是第二电源输入，它可以接可充电电池。**VBAT**是第三电源输入，接一次性电池。

DS1305的电源配置方式有三种，而且也只有三种，要对这些电源进行正确地配置，这对芯片正常工作很重要。图7-8是DS1305的一个电源配置图，一个主直流电源连接到**VCC1**，一个一次性电池连到**VBAT**（为了使图简单一些，只显示电源引脚，具体的数据接口将在后面看到）。在这个配置中，**VCC2**没有用到，必须接地。当主电源的电压降到低于电压阈值时（主电源供电失败），DS1305的内部存储器和寄存器就变为写保护，以防止被已经出了问题的微处理器破坏。

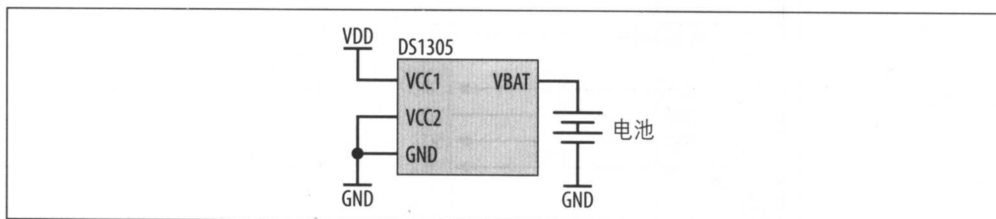


图7-8：带有一次性电池供电的DS1305

如果第二电源是一个可充电电池，那么DS1305将会如图7-9所示那样连接。当**VCC2**使用一个可充电电池时，**VBAT**必须接地。在这种模式下，如果**VCC1**供电失败，芯片没有自动写保护功能。

最后，如图7-10所示，DS1305还可以只使用一个电池作为其主电源，而没有备用电源。在这种配置中，**VCC1**和**VBAT**都接地，而电源连到了**VCC2**。

DS1305使用很简单，图7-11所示是DS1305和微控制器连接的电路原理图。

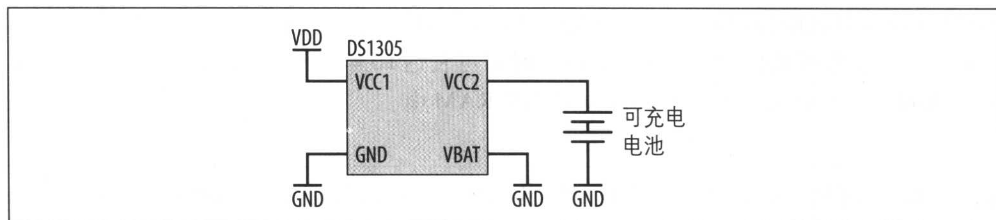


图 7-9：带有可充电电池供电的 DS1305

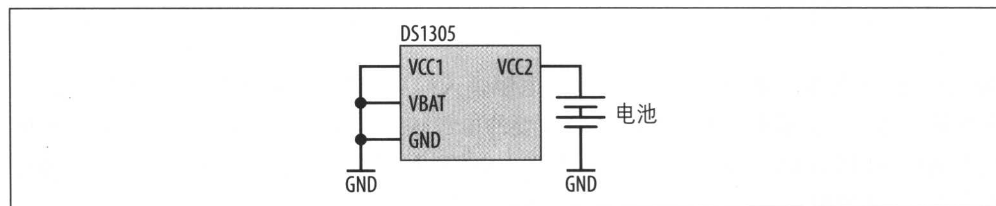


图 7-10：只用一个电池作电源的 DS1305

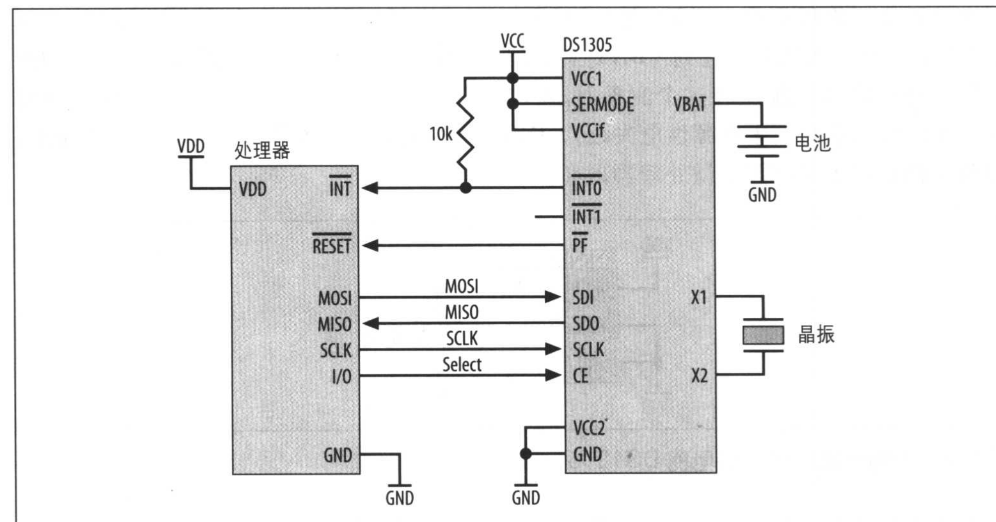


图 7-11：DS1305 实时时钟和微控制器连接

DS1305 的串行接口既可以作为 SPI 端口工作，也可以作为三线（Three-Wire）（注 1）

注 1： 由国家半导体公司开发的 Three-Wire，也叫作 MicroWire，与 SPI 非常相似，可以在许多微控制器和 DSP 处理器里找到。而与 SPI 不同之处在于，其读、写数据线是分离的，Three-Wire 使用了普通的双向数据线。

端口工作。输入**SERMODE**（串行模式）选择使用哪种模式。将**SERMODE**连到电源，就选择使用SPI工作模式；而将它接地，则选择三－线工作模式（对于三线工作模式，SDO和SDI连在一起）。将DS1305和微控制器的SPI端口相连很容易，就如同前面所说的那样，使用**MOSI**、**MISO**、**SCLK**和片选就可以实现。DS1305只有一点不同：它有一个高电平有效的**CE**（芯片使能）引脚，而不像其他常见的SPI设备那样使用低电平有效的片选引脚。因此，连接**CE**引脚的处理器I/O线应在选中该芯片时为高电平，而不选中时为低电平。

DS1305有特殊的供电失败（$\overline{\mathbf{PF}}$）输出端，当主电源**VCC1**的电压低于第二电源（**VCC2**或者**VBAT**）时，就触发这个引脚变为低电平。可以利用这个引脚来向处理器报警供电失败（作为一个中断信号），或者停止处理器工作（通过将其连到处理器的复位端）。这样做是为了防止出错的处理器在供电失败后破坏外设。如果不需要供电失败提醒，$\overline{\mathbf{PF}}$可以不连接。

输入端**VCCif**（接口电平逻辑的**VCC**）选择**SDO**和$\overline{\mathbf{PF}}$的输出电平电压。由于DS1305既可以用在5V的系统里，也可以用在3.3V的系统里，因此这个输入端可以将这些输出引脚的输出高电平设置为适当的电压。**VCCif**只是连到系统的电源上。所以，对于5V的系统，**VCCif**是5V，DS1305的输出电压也是5V；同样，对于3.3V的系统，**VCCif**是3V，DS1305的输出电压也是3.3V。

DS1305有两个中断输出：$\overline{\mathbf{INT0}}$和$\overline{\mathbf{INT1}}$。在DS1305的一个报警功能被触发的时候，这两个输出可以用来中断处理器。由于这两个中断输出是漏极开路（open-drain），因此在它们无效的时候，分别需要用一个10k欧姆的电阻将其电压提升。如果不需要中断，就不连接它们。在我们的例子中，只使用$\overline{\mathbf{INT0}}$，因此$\overline{\mathbf{INT1}}$就可以忽略不管。

最后要说的是，DS1305有两个石英晶体输入端：X1和X2。一个32.768kHz的石英晶体连接这两个引脚，用来为芯片的内部时钟提供计时源。

DS1305就是这样一个通用的小芯片，它能够为你的嵌入式系统提供计时功能。这个芯片使用简单，关于对它编程的信息在其技术手册中有详细说明。

基于SPI的数字电位计

下面让我们看一看另外一个SPI的简单例子。这次，是将数字电位计和微处理器连接起来。我们已经在第4章了解过模拟电位计。

目前，标准的电位计是手动调节的。它们或者连着旋钮（如音量控制或亮度调节），或者有一个小凹槽，可以用螺丝刀来调整。如果你的微处理器可以在软件控制下调整模拟

电路中的电位计岂不是很棒？这样，你的应用程序可以调节显示亮度或者声音系统的音量。使用一个数字电位计，你就可以做到。带有嵌入式控制器的电视机、计算机显示器和立体声设备使用数字电位计就可以调整设置，比如音量等。当你按下一个遥控器的音量按钮时，电视机或立体声设备就对数字电位计进行调节，而这个电位计正是驱动扬声器的放大器的一部分。

图 7-12 所示是一个带有 SPI 接口的 Analog Devices AD5203 数字电位计。这个芯片有 4 个电位计，每个都可以在软件控制下进行调节。每个电位计都有 64 个不同的值，不同款型的数字电位计的阻抗也不同，有 10kΩ 的，也有 100kΩ 的。引脚兼容的 AD8403 的分辨率更高，它有 256 个不同的值，也可以通过 SPI 接口进行设置。

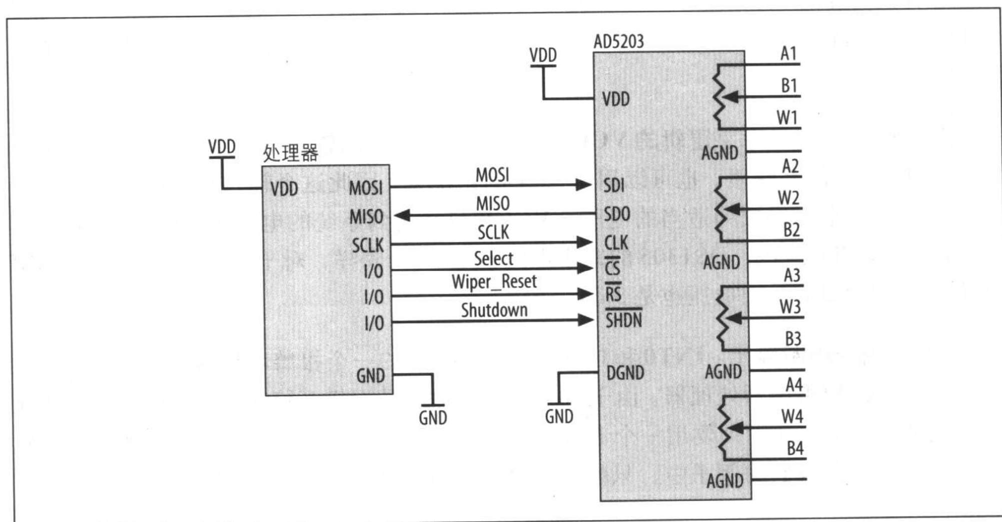


图 7-12：用 SPI 接口连接数字电位计和处理器

AD5203 有一个串行数据输入端（**SDI**），这个引脚和处理器的 **MOSI** 输出相连。与此类似，它的串行数据输出端（**SDO**）和处理器的 **MISO** 相连。AD5203 的时钟输入（**CLK**）在每个 SPI 周期中间的正缘触发，因此任何和它通信的处理器在 **SCLK** 端都必须采用时钟极性为高电平和时钟相位 1 的时序逻辑。AD5203 的片选端（**$\overline{\text{CS}}$**）可以由处理器的数字 I/O 线来驱动。这个芯片另外还有两个输入端：关闭（**$\overline{\text{SHDN}}$**）和复位（**$\overline{\text{RS}}$**）。**$\overline{\text{SHDN}}$** 将芯片置为低功耗模式，而 **$\overline{\text{RS}}$** 将电位器的滑动片置到中间位置。这两个输入端也都可以通过处理器的 I/O 线驱动，如果不需要它们发挥作用，只需通过一个 10kΩ 的上拉电阻将它们接到高电平即可。

AD5203 中的电位计的使用方法和其他电位器一样，**A** 和 **B** 两端分别连着内部电阻的两端，滑动片（**W**）的位置可在软件控制下进行调节。

AD5203 有几个接地端：**DGND** 是 SPI 接口和芯片控制逻辑的数字地；**AGND** 是内部电位器的模拟地，这两个接地端都应该在一个信号点接到 **DGND**。

AD5203 的技术手册提供了设置芯片所需的控制码，使用起来非常简单直观。

用 SPI 接口添加一个非易失性存储器

微控制器的内部存储器很小，它们的数据存储容量极其有限，现在我们就来看看如何使用 SPI 接口来添加一个 Atmel 公司的 AT45DB161 2M 串行数据闪存，以提高嵌入式系统的存储能力。这些芯片常用于低成本数码相机和应答机。你也可以将该闪存芯片用作嵌入式系统中的虚拟磁盘。

大部分其他的闪存芯片都有总线接口，但 AT45DB161 有一个串行接口，这使得它更适用于小型微控制器。AT45DB161是一个2M的闪存芯片，不过你还可以买到类似的Atmel 系列的芯片，存储容量从 512K~8M 不等。它们使用相同的物理接口，因此同样的设计对所有的芯片都适用（不过要注意，它们的引脚和外形包装有所不同，因此一个芯片不能安装到为另一个芯片设计的电路板上）。

这个芯片由一组闪存组成，按照独立的页面来组织，每页 528 字节，还有两个 RAM 缓存，每个也是 528 字节（如图 7-13 所示）。向主闪存组中写数据的时候，处理器必须先把数据写到某个缓存，然后再发命令将缓存里面的数据写入闪存。处理器既可以读缓存，将闪存页面的内容传送到缓存，也可以直接读取闪存。对两个缓存的操作是独立的，一个缓存中的内容写入闪存的时候，处理器可以（通过 SPI 接口）访问另外一个缓存。

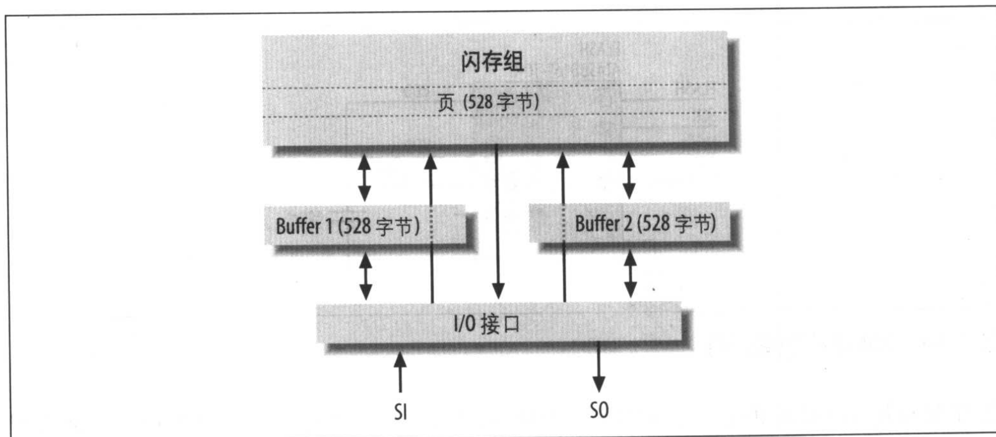


图 7-13：AT45DB161 内部结构

闪存芯片支持各种命令对缓存进行读写，将缓存里面的内容写入主闪存组，将一个闪存页面里面的数据传送回缓存。Atmel公司的技术手册里有软件协议和命令集的详细内容。

关于闪存芯片的内部结构和闪存组的有些方面需要注意。首先，528 字节的闪存页面之间并不是连续的，也就是说，如果你在软件中用指针指向存储器的当前单元，不能认为一个页面的最后一个单元地址再增加一个单元地址就是下一个页面的起始地址。每读完 528 字节（这个数字不太常见）之后，必须跳到下一个页面。可以把它想象成每个页面 528 字节，而页面之间还有很大的空间。

其次要注意的是，这个存储器的每个页面只能写1000次。大多数闪存技术（有几种不同的类型）都支持写 100000 次或更多，实际上超过这个次数存储器还能可靠地工作。然而AT45DB161不是这样的，一旦写的次数超过1000次，存储器就会出问题，它可以正确地对已经写进去的数据进行读取，但对新页面的写操作会失败。对于有的应用来说，这也许并不是什么问题。我的公司就在我们长期使用的数据记录仪中使用了这种芯片，这种设备就是为常年使用而配置的，它收集（和压缩）数据，并把它们存储在闪存芯片里。记录仪先将数据收集到缓存，直到形成一整个页面，再将数据一次写入闪存组。由于配置一次，一个页面只被写入一次（然后就移到下一个页面），因此写1000次的限制在这里不会造成问题，要配置1000次，闪存才会失效。但是如果你是用此芯片来存储变量的，并且对闪存页面中的数据进行修改是逐字节进行的，那就有麻烦了。对一个页面中的528个字节的数据逐字节单独改变需要528次，这样只做两次就会超出芯片的写次数限制。因此，这种闪存对有些应用很适合，但对其他的应用却不合适。

图 7-14 给出了 AT45DB161 的基本使用方法。

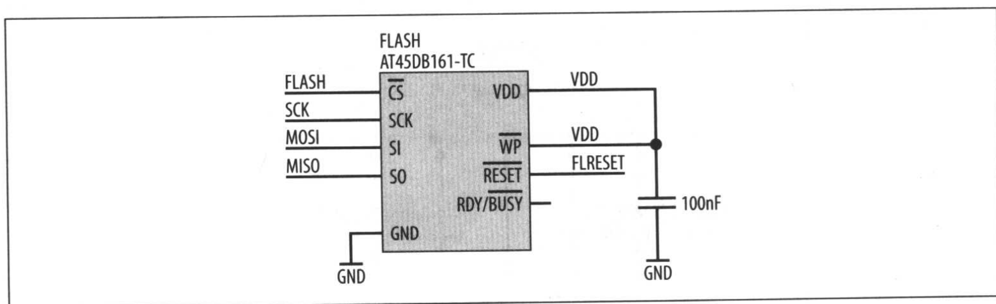


图 7-14：2M 串行数据闪存

芯片左侧是 SPI 接口的连接：**MOSI**、**MISO**、**SCK** 和一个片选（**FLASH**）。它支持的 SPI 传输高达 20MHz，因此 SPI 接口的传输速度可以非常快。芯片右侧是电源 **VDD**，它通过一个 100nF 的电容接地进行退耦。AT45DB161的电源电压要求在2.5V~3.6V之间，

它的逻辑输入端可接收的电压是5V，这意味着该芯片可以用在混和电源系统中，也就是说，当它需要一个3V的电源时，可以直接连到一个电源是5V（逻辑高电平也是5V）的处理器。AT45DB161 有个写保护引脚（**WP**），当这个引脚为低电平时，就禁止对闪存中的内容进行修改，如果不需要此功能，就可以直接将它接到高电平。该闪存芯片还有一个 **RESET** 输入端，因此可以在软件控制下进行手动复位。芯片还内置有一个开机复位功能，可以把芯片设为一个特定的状态，因此手动复位就没有什么必要了。但是我发现内部的开机复位发生器对条件的要求似乎过于苛刻，不能每次都发挥作用。如果开机复位发生器不能发挥作用，芯片就不能进入预定的状态，也就无法被系统使用。因此，我认为将闪存复位的控制权交给处理器，作为处理器初始化的一部分是个很不错的做法。当执行处理器的复位固件对处理器进行初始化时，也将闪存芯片复位。这是个很简单的事情，但对于一个可靠的系统来说就非常重要。引脚1是一个状态输出端（**RDY/BUSY**），用来告诉你芯片是空闲的还是正在完成内部操作。

图 7-15 是这个存储器芯片和一个 Atmel 公司的 90S4434 AVR 处理器的连接示意图。图中的AVR部分和我们前面看过的没什么不同，这也是诸如SPI这样的简单接口的一个优点，它们将小的子系统模块连接到一起，就像搭积木一样。从基本的核心设计开始，然后按你的需要添加外设。图中还给出了电源的退耦电容、处理器的石英振荡器和**RESET**端的上拉电阻。引脚 41（**PB1**）是闪存的一个“手动”（由处理器控制的）复位输入。

使用 SPI 接口添加一个参数存储器

在前一节，我们了解了如何添加一个大容量的串行闪存来存储数据。在系统没有通电的时候，一些重要的关键变量需要保存，用非易失性存储器来保存这些系统参数很有价值。但 AT45DB161 数据闪存不是用来完成这个功能的，它更适合于数据记录，而且它的容量对于参数存储来说太大了。因此，现在我们来看一下如何使用 SPI 在嵌入式系统中添加一个小参数存储器（EEPROM 类型）。我选择的 EEPROM 是 Atmel 公司的 AT25640，这种芯片在不通电的情况下数据可以保存至少 100 年，而且可写 100 万次以上（大大超过AT45DB161!）。这样，你的软件就可以随意改变参数变量而不用担心芯片使用寿命的限制。AT25640 只有 8K 的存储容量，这个数字听起来可能不是很大，但是不用担心，因为 8K 也就是 8192 个字符型变量，已经远远超过了存储大多数参数需要的存储空间。如果 8K 太大了，还有其他容量为 1KB（AT25080）、2K（AT25160）和 4K（AT25320）的存储器。

AT25640的结构和使用方法比AT45DB161要简单得多，在Atmel公司的芯片的技术手册中附有软件协议的细节。

图 7-16 所示为 AT25640 的电路原理图。

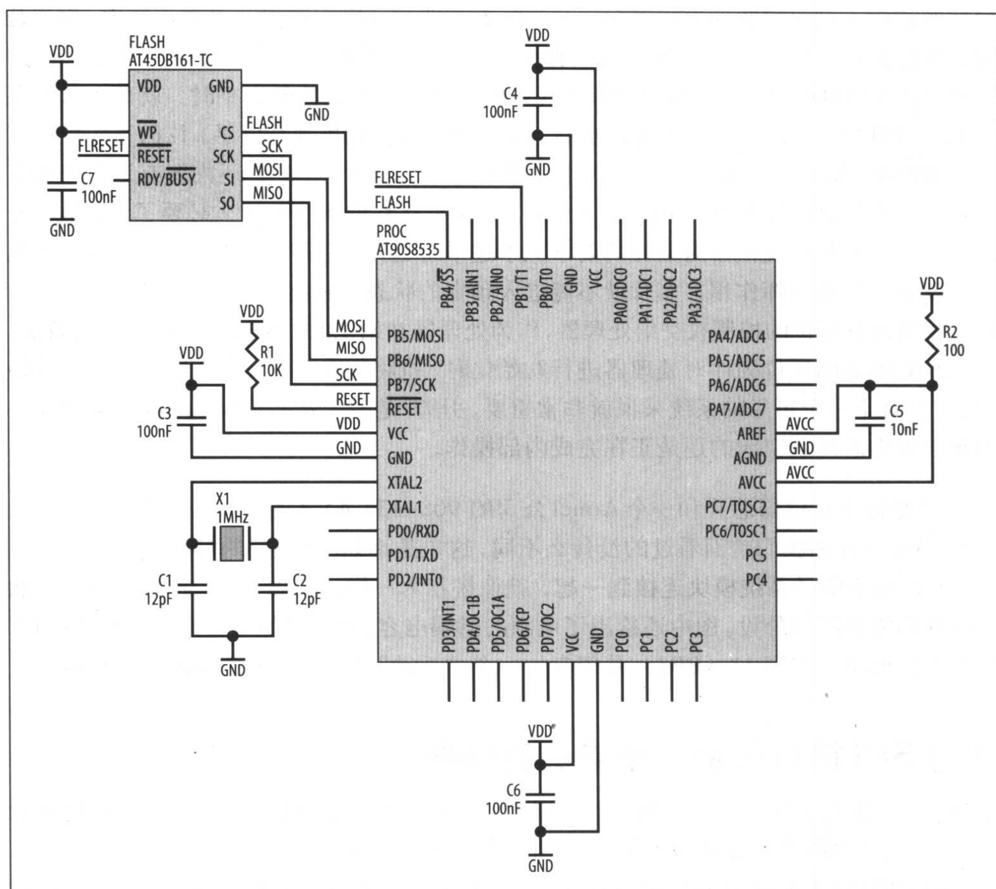


图 7-15：一个 2M 数据闪存和 AT90S4434 的连接

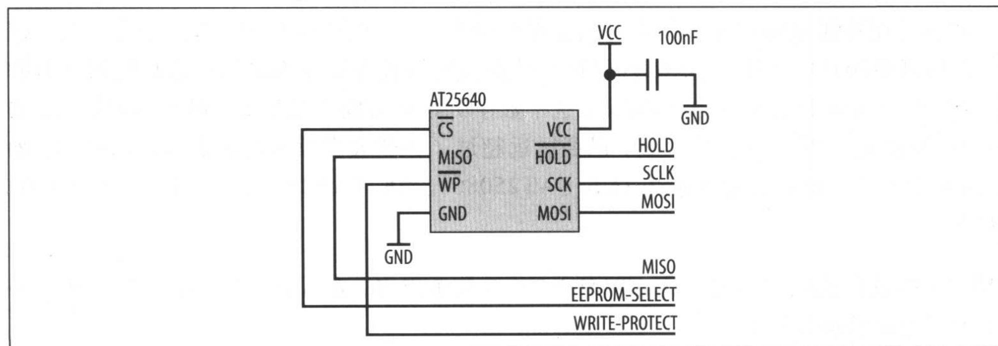


图 7-16：使用 AT25640 EEPROM

所使用的接口是标准SPI接口，芯片还有一个写保护输入端和一个保持输入端。保持端有效允许处理器可以暂时终止一个串行传输（而去做其他的事情）而不停止AT25640。如你所料，写保护端有效的时候，存储芯片就变为只读设备。这些控制输入端可以由处理器的可编程I/O线来驱动。该芯片还要求的就是电源（通过一个100nF的电容接地退耦）和接地。AT25640有两种类型，其中一种的工作电源电压是2.7V~5.5V，而另一种的电源电压是1.8V~3.6V。

第8章

用 I²C 添加外部设备

无一分可增不是完美，无一分可减才叫完美。

——安东·德·圣艾修伯里

在上一章，我们了解了用来连接外设芯片和微控制器的低成本SPI接口。在这一章，我们将学习另一种用来连接外部设备的串行接口：I²C。

I²C 简介

I²C（Inter Integrated Circuit，内部集成电路）总线是价格低廉却很有效的用以互连小规模嵌入式系统内的外设的网络。I²C总线有时候也叫作IIC，它已有20多年的历史了。I²C接口和SPI接口的作用相同，但二者的使用方法有些不同。

I²C总线用两根线来连接多支路总线中的多个设备。这种总线是双向、低速的，并与公共时钟同步。可以直接将一个设备接到I²C总线上或是从该总线上取下，而不会影响其他设备。一些生产商比如Microchip公司、Philips公司、Intel公司等生产的小型微处理器都内置了I²C接口。I²C总线的数据传输率比SPI总线要慢一些，在标准模式下的传输速度为100kbps，在快速模式下为400kbps。

利用I²C接口在设备之间进行连接使用的两根线是SDA（串行数据）和SCL（串行时钟），它们都是开漏（注1），通过一个上拉电阻接到正电源，因此在不使用的时候仍保

注1：　开漏或开路集极引脚具有输出驱动，能够把信号线置为接地，但不能够把信号线提升至高电平。因而具有这样的优势：连接于信号线的不止一个的设备可以使其降至低电平。若非如此，如果一个设备试图把信号线置于低电平而另外一个设备却想把它置为高电平，那么就会产生短路，导致严重的后果。当处于空闲状态（没有设备）时，就由电阻将其置为高电平。

持高电平。使用 I^2C 总线进行通信的设备驱动这两根线变为低电平，在不使用的时候就让它们保持高电平。每个连到 I^2C 的设备都有一个唯一地址，这个设备可以是数据发送者（总线主机）、接收者（总线从机），也可以二者都是（如图 8-1 所示）。I^2C 是多主机总线，这意味着可以有多个设备充当总线主机的角色。

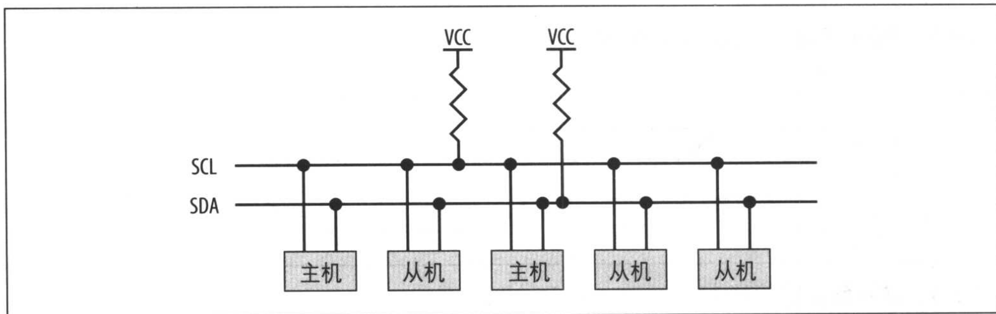


图 8-1：I^2C 总线框图

SDA 和 **SCL** 都是双向的。SPI 总线有两根单独的线，分别用于两个方向的通信，而 I^2C 总线不同，它使用同一根线来完成主机发送数据和接收从机响应。另外，与 SPI 总线具有多个工作模式不同，I^2C 总线只有一个工作模式，时钟、**SCL** 和数据线 **SDA** 之间的时序关系很简单直观：当空闲的时候，**SDA** 和 **SCL** 都是高电平，只有 **SDA** 变为低电平，接着 SCL 也变为低电平时，才开始 I^2C 总线的数据传输（如图 8-2 所示）。当 **SDA** 和 **SCL** 都变为低电平时，就是告诉总线上的所有接收设备数据包的传输开始了，在 **SCL** 变为低电平后，**SDA** 才发送（高电平或者低电平）第一个有效数据位，这被称为开始条件。

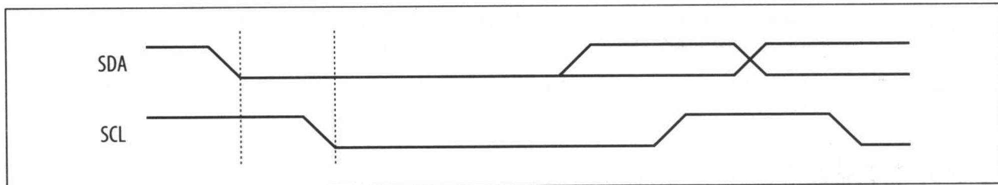


图 8-2：数据包传输的开始

对于被传输的每一位，当 **SCL** 为低电平时在 **SDA** 上必须变为有效，该位是在 **SCL** 的上升沿对 **SDA** 上的数据位进行采样的，也必须一直保持有效直到 **SCL** 再次变为低电平，然后 **SDA** 就在 **SCL** 再次变为高电平之前传输下一位（如图 8-3 所示）。

最后，**SCL** 变回高电平（无效），接着 **SDA** 也变为高电平（如图 8-4 所示），数据传输结束。这被称为结束条件。

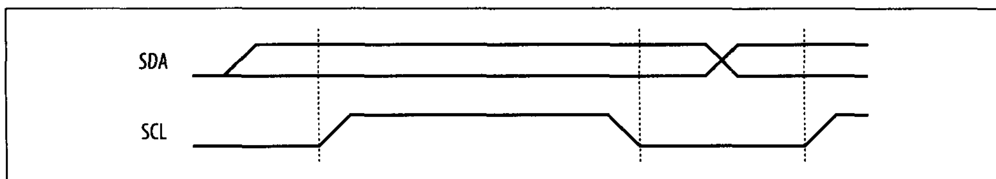


图 8-3：SDA 和 SCL 之间的时序关系

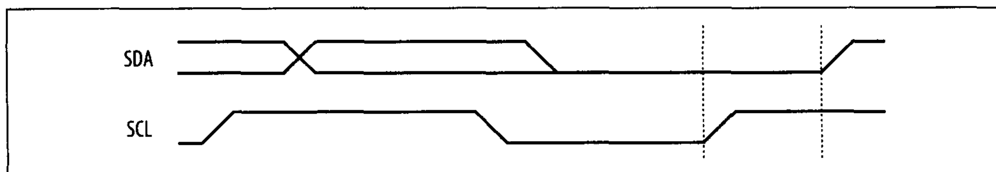


图 8-4：数据包传输的结束

无论多大的数据包都可以通过 I^2C 总线进行传输。像 SPI 总线一样，I^2C 也是高位先传输。如果数据接收者无法再接收更多的数据，它可以通过将 **SCL** 保持低电平来中断传输，这样可以迫使数据发送者等待，直到 **SCL** 被重新释放。

发送方发出的每个字节都必须经过接收方确认，每个字节的第 8 位数据一旦传送结束，发送方就释放数据线 **SDA**，然后主机在 **SCL** 上产生一个额外的时钟脉冲，这会触发接收方通过将 **SDA** 置为低电平来表示对接收到的字节进行确认（如图 8-5 所示）。如果接收方没有能将 **SDA** 置为低电平，发送方就会中断传输，并且采取适当的错误处理措施。

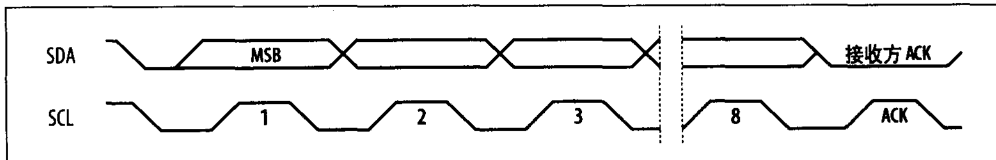


图 8-5：带有接收方确认的 I^2C 数据包

现在，I^2C 是多主机总线。因此，存在同一时间会有多个主机试图开始数据传输的可能。由于 I^2C 总线默认的状态是高电平，所以一个主机发送一个数据位 0 时会将 **SDA** 置为低电平，而如果这个数据位是 1 时就将总线置为默认状态。所以若两个主机要同时传输数据，一个主机发送数据位 1 将总线置为默认状态，但又检测到总线被另外一个主机置为了低电平（发送数据位 0），该主机将记录一个错误状态，并且终止数据的传输。

SPI 总线使用一个独立的片选端来使接收从机有效，每个 SPI 从机都有一个单独的片选端，由主机驱动。I^2C 没有这样的选择机制，不过总线上的每个设备都有一个唯一的地

址，数据包传输时先发送地址位，接着才是数据。一个地址字节由7个地址位和1个指示位组成。如果指示位是0，意味着这个传输是一个写操作，被选中的从机将接收数据并将其作为输入；如果指示位是1，就要求从机将数据发送回主机。图8-6所示是一个1字节数据包的示意图。

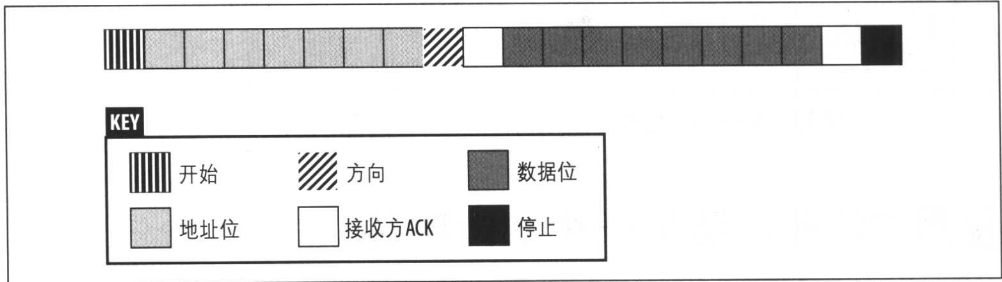


图8-6：一个I²C总线数据包

有个特殊的地址，称为全呼叫地址（general call address），用于向所有的I²C总线设备进行广播。这个地址由%0000000和一个方向位0组成。全呼叫是一种使主机可以确定哪些从机有效的机制，它有几种类型。一个全呼叫的第2个字节说明呼叫从机的目的，一旦接收到第2个字节，每个从机就会检查这个指令对自己是否有效，如果有效就会向主机进行确认，否则就不予理会。如果全呼叫的第2个字节是0x06（%00000110），意思就是让有效的从机进行复位并将它们的地址告诉主机；如果第2个字节是0x04（%00000100），相应的从机就只需将它们的地址告诉主机，而不必复位；当这个字节为其他内容（最低位为0）的时候，从机可以对这个全呼叫不予理睬。

如果第2个字节的最后一位是1，这个全呼叫就是一个主机设备通过向系统中其他主机发送自己的地址来声明自己，这个字节的其他位包含主机地址。

另外一个特殊的地址字节称为开始字节，这个字节是%00000001（0x01），它用来告诉其他主机一次大规模的数据发送即将开始，这点对于没有专用I²C硬件而又必须通过软件轮询来对I²C总线进行监听的主机来说尤为重要。当一个主机探测到另外一个主机产生的开始字节时，它就会降低自己对总线的轮询频率，为其他的软件任务提供更多的时间。

I²C总线还支持一个扩展的10位寻址模式，可连接的外设数量可达1024个，使用7位寻址模式的设备和10位寻址模式的设备可以在同一个系统中混合使用。10位寻址时，使用2个字节来保存地址。如果（第1个）地址字节以%11110XX开始，就会产生一个10位地址，第1个字节的最低两位和第2个字节的8位合起来构成10位的地址（如图8-7所示）。而7位设备将会忽略这个过程。

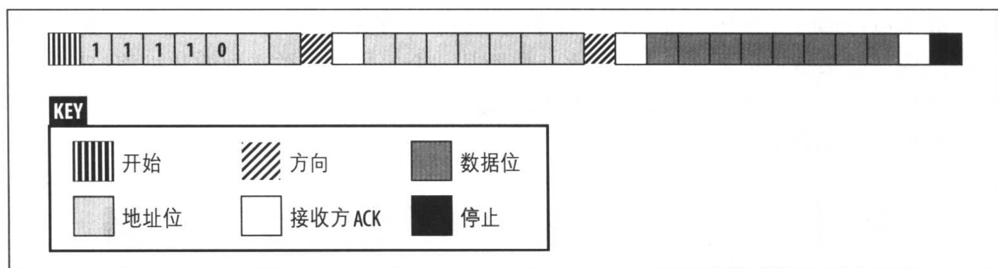


图 8-7：一个 10 位寻址的 I²C 数据包

使用 I²C 接口添加一个实时时钟

在前一章，我们已经知道了如何使用 SPI 接口将实时时钟（RTC）和微处理器相连，现在再来看看如何用 I²C 接口来完成同样的工作。在这个例子中要使用的是 Philips 公司的 PCF8583。它的体积极小，而且像 DS1305 一样带有 240 字节的 RAM，可以用来保存参数。与 DS1305 不同的是，PCF8583 没有集成电池备份系统，因此需要你自己提供一个外部的电池备份电路。还有很多其他 I²C 总线类型的 RTC，有些也内置了电池失效保护功能，我之所以选 PCF8583 是因为它是 I²C 接口中一个非常简单的例子。

PCF8583 有 2 个引脚（**OSCI** 和 **OSCO**），用以连接一个 32.768kHz 的时钟石英，这个石英为一个充当计时功能的内部电路提供脉冲。地址引脚 **A0** 决定这个设备在 I²C 总线上的地址，大多数 I²C 芯片都提供几个地址引脚，以使设备拥有一个范围内的几个地址。然而 PCF8583 为了减少引脚的数目，只有一个地址引脚，它的 6 个地址位在内部通过硬布线的方式已经确定，只有最低位 **A0** 可以供系统设计者使用。图 8-8 所示是 PCF8583 的地址配置示意图，注意数据传输指示位（读或写）是如何合并到地址域中的。

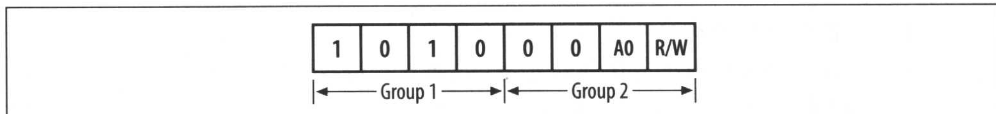


图 8-8：PCF8583 地址

引脚 A0 直接接地就将相应的地址位设为 `0`，这样 PCF8583 在 I²C 总线中的映射地址就为 `0x50`。或者，如果 **A0** 连到 **VDD**，则设备的地址就是 `0x51`。

图 8-9 所示是 PCF8583 和一个微控制器相连的示意图。

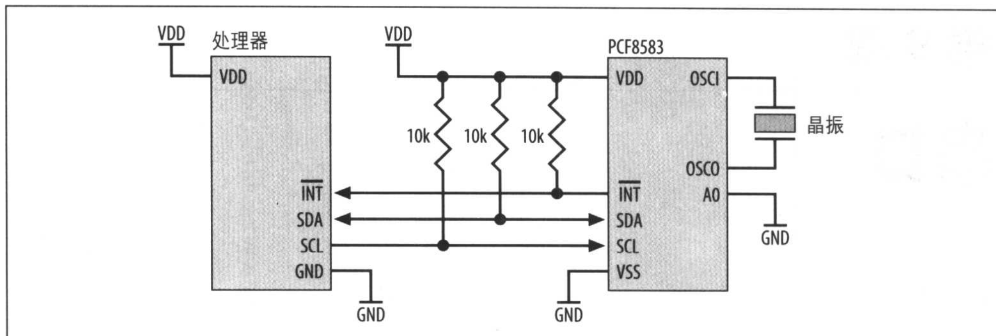


图 8-9：PCF8583 和微控制器的连接

SDA 和 **SCL** 都需要通过一个上拉电阻连到 **VDD**。PCF8583 还有一个内部报警功能，可以产生一个输出（**INT**）来中断处理器，由于这个输出是开漏，因此还需要一个上拉电阻。

使用 I²C 接口添加一个小显示设备

可以使用I²C接口向嵌入式计算机添加简单的LCD（以及其他有同样功能的显示部件），这些LCD通常只有几行亮的文本，但在显示简单信息的时候却非常有用。Matrix Orbital公司（*http://www.matrix orbital.com*）生产许多种显示模块，比如VFD2041，而且很容易使用。VFD2041 可以显示 4 行，每行 80 个字符，图 8-10 所示是它的接口电路图。可以看到，基本上没什么东西。而台式机上的LCD相比之下要复杂得多，将它们和小处理器连接是不明智的选择。对于简单信息（比如在一个仪器的面板上）的显示来说，图 8-10 所示的电路就已经很理想了。

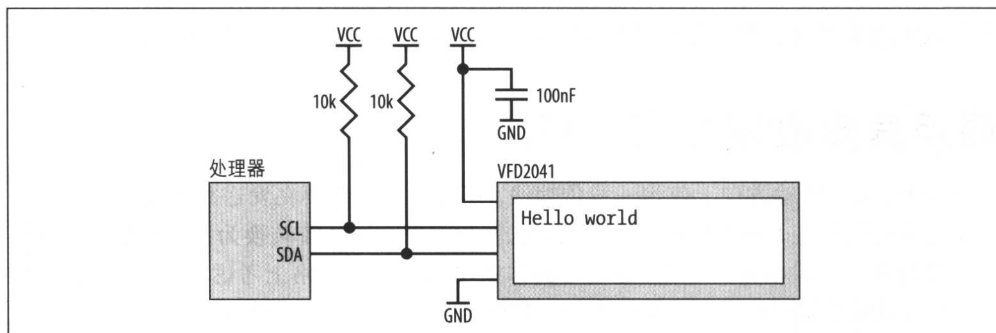


图 8-10：使用 I²C 接口连接 VFD2041

Matrix Orbital 公司的许多显示器都支持 RS-232C 接口，因此即使你的嵌入式处理器不支持 I²C 接口，仍然可以很容易地添加一个小型显示器。

第9章

串口

我全部的经历，也只是一座拱门，
尚未游历的世界在那门外闪光，
随着我一步一步地前进，
它的边界也不断地向后退逝。

——阿尔弗莱德·丁尼生
《尤里西斯》

在本章中，我们将要了解嵌入式系统如何通过无所不在的串口与外面的世界连接起来。首先，我们会看到如何实现一个典型的RS-232C串口，以及如何透过RS-232C端口来为嵌入式系统供电。继而，我们会了解较稳固的RS-422，此串口专门针对远距离快速传输数据而设计。最后，我们会学习RS-485，这一串口是对RS-422的扩展，主要是针对嵌入式计算机的低成本网络而设计。

现在从驱动串行接口的引擎通用异步收发器（UART）来正式开始进行我们的串口介绍。

通用异步收发器（UART）

串行I/O在每个传输方向上使用一根传输线。所有的串口都要在发送数据端将并行数据转换为串行比特流，而在接收端反过来，也就是将串行比特流转换为并行数据。当系统之间进行并行传输数据——无论是在具体实现上还是从价格上考虑——都不太实用时，就可以采取串行通信。这样的串行通信可能发生在计算机和终端之间，也可能发生在计算机与打印机、掌上电脑或远程控制的红外设备之间，还可能是更高级的形式——高速网络通信，比如以太网。对于嵌入式计算机来说，连接到一台主机采用简单的串口是最容易也是最廉价的方式，这个串口可能是应用的一部分，也可能只是作为调试之用。

串口最简单的形式是通用异步收发器（Universal Asynchronous Receiver Transmitter，UART），有时候也叫作异步通信接口适配器（Asynchronous Communication Interface Adapter，ACIA）。它们之所以都称为“异步的”，是因为时钟信号没有与串行数据一起传输，接收者必须时刻关注数据，对每个位进行探测，而不是花费一个时钟周期来达到同步。

图 9-1 所示是一个 UART 的功能模块图，它由两部分组成：一个将串行比特流转换为微处理器可以使用的并行数据的接收器（Rx），和一个将来自微处理器的并行数据转换为串行形式进行发送的发送器（Tx）。UART 还提供一些状态信息，比如接收器是否已满（有数据到达）或者发送器是否为空（有数据待发送）。许多微处理器芯片都内置了片上 UART，但对于大型系统来说，UART 通常是一个独立的设备。

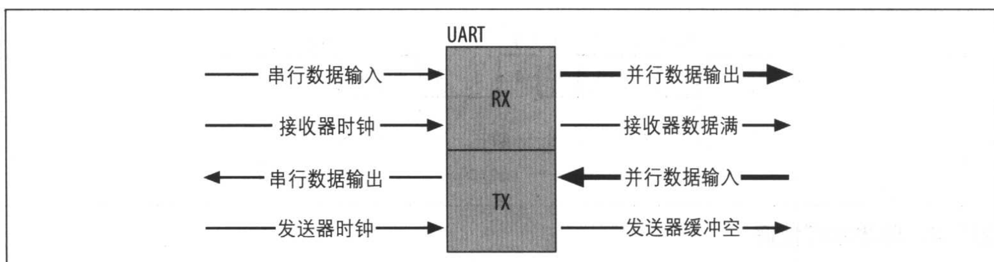


图 9-1：通用异步收发器功能框图

串行设备每次发送一个比特的数据，因此常规的“并行”数据在发送之前必须先要转换为串行形式。UART 实际上就是一个额外增加了一些特性的并－串行转换器。UART 发送器本质上是一个移位寄存器，可以并行装载数据，然后在串行时钟脉冲的控制下再将数据一位一位地按顺序移出。反过来，接收器是把串行比特流接收到一个移位寄存器中，然后由处理器并行读取。

注意： UART 实际上比基于半导体的计算机出现得还要早。在电子通信的早期，UART 是一个由齿轮、继电器和电动机械寄存器组成的机械设备，如果要对它进行设置，首先就需要有一个扳手！

串行传输时涉及到的问题之一是在接收端对数据进行重组，其困难在于对比特之间界限的探测。比如，如果串行线保持一定时间的低电平，接收数据的设备必须能够识别这个比特流代表的是 00 还是 000。它必须知道一个比特在什么地方结束，下一个比特从什么地方开始。发送设备和接收设备可以通过共享一个公共时钟来做到这一点。因此，在同步串行系统中，串行数据流通过与数据流一起传输的时钟信号进行同步，这样简化了对

数据的校正，但却需要增加一根信号线来传输串行时钟信号。异步串行设备（比如UART）则不共享公共时钟，而是每个设备都有自己的本地时钟，这些设备必须严格地在同一频率下工作，而且还需要添加逻辑电路来探测所传输数据的相位，以及为其对接收端时钟进行锁相。

异步传输用于那些每次发送一个字符，并且传输时各个字节之间的时间间隙可能发生变化的系统中。数据格式是字符的第1位作为起始位，最后的1或2位作为停止位（如图9-2所示）。接收器在收到起始位时对其时钟进行同步，接着对数据位（第7或8位，视系统设置而定）进行采样。当以正确顺序接收到停止位序列时，接收器就认为传输结束，已经得到一个有效的字符。如果没有接收到适当的停止位序列，接收器就认为是它的时钟与时钟周期不符，将会声明一个帧错误或位偏差错误。

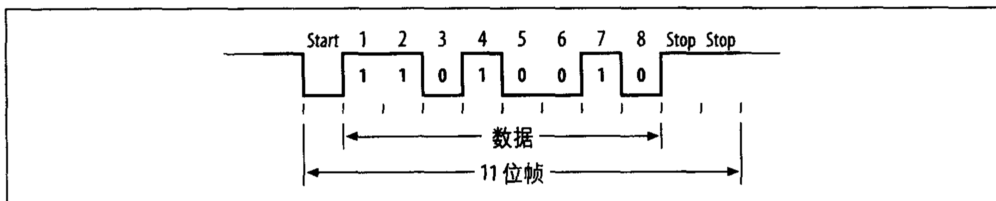


图9-2：异步串行数据

数据从并行到串行格式的转换通常由专用的UART硬件来完成，但在只有并行I/O的系统中，这个转换可能是由软件实现的，它锁定并行I/O端口的1个位作为串行线使用。

错误检测

在任何存在着潜在噪声的介质（比如串行线）中进行数据传输时，都有可能出现错误。为了检测这样的错误，许多串行系统采样奇偶校验来对数据有效性进行简单的检查。一个字节的校验位由发送方UART计算，包含在该字节中作为传输数据的一部分。接收方UART也对该字节的校验位进行计算，并和接收到的校验位进行比较。如果一致，接收器就认为一切都正常；否则，就可以确定某个（些）地方出现了错误。

奇偶校验有几种类型，主要的两种是奇校验和偶校验。任何数据字节中1的个数不是偶数就是奇数，向这个字节再添加1位（奇偶校验位）来使得1的个数为偶数（偶校验）或者奇数（奇校验）。在正确的数据传输中，接收器和发送器必须设置为相同的奇偶校验类型。由于没有在UART之间建立公共奇偶校验的协议，因此必须在每一端进行手工设置。

因此，对于二进制序列`%01000000`，奇偶校验位采取偶校验时应该为1，采取奇校验时为0。同样，对于`%11111111`，采取偶校验时奇偶校验位为0，而采取奇校验时应为1。

奇偶校验的计算和检测是通过UART中特定的硬件自动完成的，这里没有什么明确需要计算的内容，但你必须确定对奇偶校验类型的设置正确，否则，设备就会不知道如何产生相应的奇偶校验信息。

在接收端检查奇偶校验位，以防止在传输过程中数据的任何一位发生变化。假设我们发送的是 `%01000000`，UART设置为偶校验，根据 `%01000000` 计算得到奇偶校验位应该为1，现在假定数据在传输过程中被破坏，实际接收到的是 `%01000001`，接收器根据这个字节计算的奇偶校验位为0，与接收到的奇偶校验位1相比较后不一致，这样就检测到一个奇偶校验错误，这时接收器将会采取一定的措施（比如请求重发该字节）。

现在，如果传输介质的干扰比较严重，有两个位发生变化将会怎么样呢？假设我们还是采取偶校验方式发送 `%01000000`（计算得奇偶校验位=1），传输过程中数据变成了 `%01001001`，接收器根据接收到的数据计算得到的奇偶校验位也是1。传输过程中数据被破坏了，但却没有检测到奇偶校验错误！可以看到，这种错误检测方法的检错能力非常有限，因此常常采用更复杂的错误检测（以及纠正）方案，其中一个比较好的例子是循环冗余校验（Cyclic Redundancy Check，CRC）算法。如果要实现CRC，网上有充足的源代码可以参考，用喜欢的搜索引擎搜一下就会知道。

前面讲述了如何串行传输比特的基本知识，现在该看看如何实现一个物理的串口。我们从历史悠久的串行连接两台计算机（或任何其他数字设备）的标准开始说起。

历史悠久而可靠的RS-232C

RS-232C是一种串行通信接口标准，自20世纪60年代开始，它就以各种不同的形式在被使用。RS-232C连接的串行设备之间的距离可达25米，传输速度可达38.4kbps，使用它可以连接其他计算机、调制解调器，甚至老式终端（在测试时监控状态信息非常有用的工具）。过去，打印机、绘图仪和其他设备的主机都是使用RS-232C接口。现在，由于需要高速传输大量数据，RS-232C作为一种连接标准正逐渐被高速网络，比如以太网取代，不过，它对于嵌入式系统来说仍然是一种非常有用的（还很重要）而且简单的连接工具。

RS-232C采用的是不平衡（unbalanced）的传输方式，也就是说传输的数据位的电压电平是相对于本地的地，它的逻辑高电平是−5V~−15V（通常为−12V），逻辑低电平是+5V~+15V（通常为+12V）。因此，要清楚RS-232C与计算机其他部分的逻辑不同，它的高电平是负电压，而低电平是正电压。

RS-232C中使用的术语也可以追溯到20世纪60年代，在那时候的大型机中，高电平（1）

称作“间隔（space）”，而低电平（0）称作“标志（mark）”，现在在关于RS-232C的一些短语比如“标志奇偶校验（mark parity）”和“间隔奇偶校验（space parity）”中仍可以看到这些术语。你还可以看到RS-232C接口的系统仍然使用7位数据帧（20世纪60年代遗留的另一个术语），而不是更常见的8位数据帧。实际上，这也是在因特网上发送的电子邮件是7位字符的原因之一，是为了防止数据包通过只支持7位传输的串行连接进行路由时出现问题。知道历史的碎片还如何地徘徊萦绕在我们的周围也是一件美妙的事！但RS-232C数据传输时更常用的还是8位字符，因此你要实现任何串口也都应该这样做。

RS-232C连接包括一个驱动器和一个比较器，如图9-3所示。

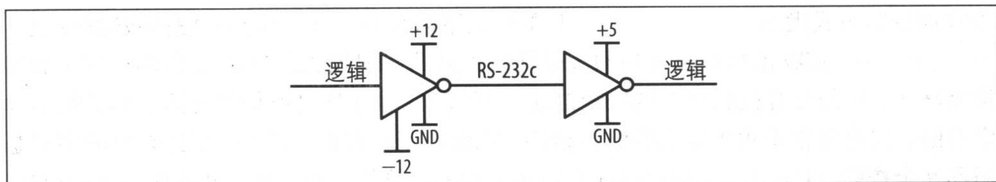


图 9-3：RS-232C

尽管还存在很大改变的可能性（因此就会有很多不兼容存在），RS-232C还是对连接器和引脚的分配进行了定义。RS-232C刚开始的目的是用来连接数据终端设备（Data Terminal Equipment, DTE）和数据通信设备（Data Communication Equipment, DCE），如图9-4所示，因此定义的标准假定RS-232C连接的一端是DTE设备，另一端是DCE设备。在出现计算机之前，DTE是一个终端或电传打字机，而DCE是一个调制解调器。调制解调器提供一个连到电话线的接口，因此可以和远处的调制解调器以及终端相连。

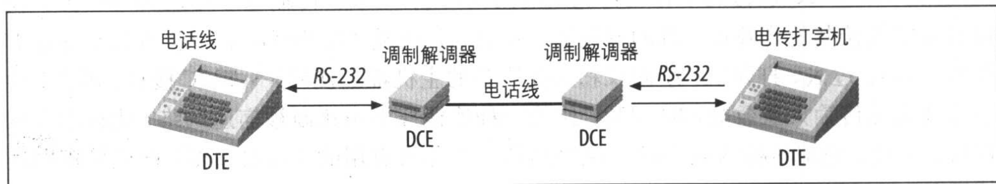


图 9-4：最初使用 RS-232C 连接电传打字机和调制解调器

在台式计算机出现之前，这种连接方式简单明了。但当你希望连接终端或调制解调器到计算机的串口时，问题就出现了。将计算机作为DTE还是作为DCE？RS-232C标准意味着如果连接一端是终端，那么另一端就应该是DCE。因此，如果是连接一个终端和Unix工作站，RS-232C标准就认为工作站是一个DCE（如图9-5所示）。反过来，如果是连接调制解调器和计算机，则计算机应该是一个DTE（如图9-6所示）。

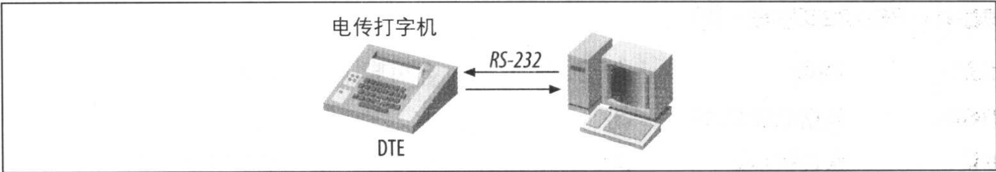


图 9-5：连接到计算机的 DTE 设备

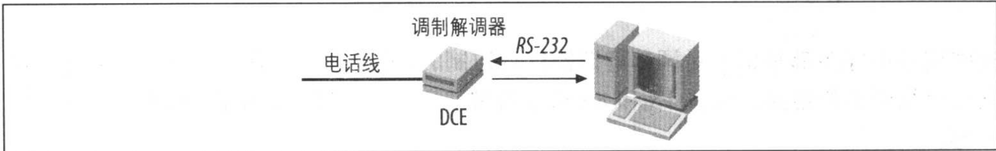


图 9-6：连接到计算机的 DCE 设备

当面对这些问题的时候，生产商们就会擅自根据他们自己的意愿来选择。IBM PC 有一个 DTE 类型的连接器，然而 Unix 工作站的生产商（比如 Sun 公司）常常选择让它们的机器带有 DCE 连接器，因此它们很可能是连接到终端的。如果要想连接一台 PC 机和一个调制解调器，需要一根 DTE-DCE 电缆；要连接一台 PC 机和一个终端，需要一根 DCE-DTE 电缆；要连接一个 Sun 工作站和一个终端，需要一根 DCE-DTE 电缆；要连接一个 Sun 工作站和一个调制解调器，需要一根 DCE-DCE 电缆；要连接两个 Sun 工作站，需要一根 DCE-DTE 类空调制解调器电缆（**Rx** 和 **Tx** 相交换）；要连接一个 Sun 工作站和一个 PC 机，需要一个 DCE-DTE 类空调制解调器电缆；而要连接两台 PC 机，就需要一根 DTE-DTE 类空调制解调器电缆。所以，仅仅对于两种设备（DCE 和 DTE），根据排列组合就需要 6 种类型的电缆！因此，就像有些人说的那样：多样性是生活的调味品，但对 RS-232C 来说却是毒药！

表 9-1 所列是 RS-232C 的标准连接，包括 25 引脚和 9 引脚的连接器。信号的名字都是和 DTE 相关的，比如 **Tx** 指数据传输是从 DTE 到 DCE 的。

表 9-1：RS-232C 信号

信号	功能	25 引脚	9 引脚	方向
Tx	发送数据	2	3	从 DTE 到 DCE
Rx	接收数据	3	2	从 DTE 到 DCE
RTS	请求发送	4	7	从 DTE 到 DCE
CTS	清除发送	5	8	从 DTE 到 DCE
DTR	数据终端就绪	20	4	从 DTE 到 DCE
DSR	数据设备就绪	6	6	从 DTE 到 DCE

表9-1：RS-232C信号（续）

信号	功能	25引脚	9引脚	方向
DCD	数据载波检测	8	1	从DTE到DCE
RI	振铃指示器	22	9	从DTE到DCE
FG	帧接地（机壳）	1	-	共用
SG	信号地	7	5	共用

这些信号中许多都是用来对调制解调器进行控制的，要在一台计算机和一个终端之间建立一个很简单的连接，仅需**Tx**和**Rx**信号就够了。另外，许多系统都将**FG**和**SG**连到一起。

握手

当两个远程系统串行通信的时候，在接收者对接收到的数据处理完之前，必须禁止发送者发送新的数据，这个过程称作握手（shake hands），或者流控（flow control），其具体实现很简单。在发送1个比特（或数据包）之后，发送者在收到接收者表示就绪的确认信息之前不会再发送数据。握手有3种可能的形式：软件方式、硬件方式和无（不进行握手）。

显然，不进行握手是最简单的方式，用于数据传输过程中发送系统在准备和发送数据上都比接收系统慢得多的场合。例如，如果是一个1MHz的低速小型嵌入式计算机发送数据给一个1GHz的高速计算机系统，就可以假设无需进行握手，高速计算机也可以胜任接收数据的任务。不过，如果高速计算机是运行在一个比较流行的操作系统（这些操作系统的一个典型特征是对实时事件反应能力弱）下，它很可能就无法正确接收到全部数据。在这种情况下，将需要进行握手，并且很可能可以通过握手很好地解决问题。但如果是通过串口为计算机提供一个手工输入接口，你完全可以认为没有人的输入速度会比计算机的处理速度还快。因此，对于只用来供使用者访问或测试的串口，可以忽略握手过程。

在RS-232C中，硬件握手使用两个信号：**RTS**（请求发送）和**CTS**（清除发送）。当发送者希望发送数据时，就置**RTS**为有效，告诉接收者有数据等待发送，而当接收者准备就绪时，它就将**CTS**置为有效，来通知发送者可以发送数据了。这样，数据传输的快慢就取决于握手速度。

软件握手又称为*XON/XOFF*，用于接收者和发送者之间无法实现硬件握手的场合，比如数据的传输是通过一根电话线来实现时。软件握手用两个字符来实现流控，一个代表请

求对方暂停传输，另一个代表清除暂停传输的请求，继续数据传送。通常情况下这两个字符是Ctrl-S（0x13）和Ctrl-Q（0x11），在传输的文件中禁止包含这两个字符，因为它们会被接收者理解为流控制字符而不是数据。假如发送的只是ASCII字符，就不会存在这样的问题，但是如果是发送二进制数据，就会遇到这种情况。常见的解决办法是在发送之前对二进制数据进行预处理，将它们用ASCII字符代替。比如，0x2F变为ASCII字符“2”（0x32）和“F”（0x46），接收端的软件会把ASCII字符变回二进制数据。Unix下的软件uuencode和Mac OS下的BinHex就是用来做这些工作的。

RS-232C接口的实现

向系统添加一个RS-232C接口很容易，大多数微控制器芯片（除了极小型的）都内置了UART，因此所需要的仅仅是一个外部电平转换器来将串行传输的数据转换为RS-232C数据和将RS-232C数据转换为串行传输的数据。Maxim公司开发了大量的RS-232C接口芯片（串行转换器），可以大大简化设计过程。无论需要进行什么类型的转换，毫无疑问都可以用Maxim公司的产品来实现。通常情况下，MAX3222收发器会是一个比较理想的选择。由于几乎所有的RS-232C收发器的使用方法都相同，因此一个使用MAX3222的设计是了解如何使用其他任何收发器的好例子。不像其他的许多电平转换器，Maxim公司的产品可以在3.0V~5.5V的低电压电源下工作，其他生产商生产的同类产品大都要求+12V和−12V的电源，因此还需要一个额外的电压调节器。MAX3222耗电最低（普通工作模式下电流为1mA，关闭模式下不超过1μA），因此使得它成为便携式和电池供电应用的理想选择。如果你不需要关闭串口进入低耗电模式，可以用MAX3232来代替。这种芯片和MAX3222相比，除了不具备低性能外其他的功能是一样的。

MAX3222的使用极其简单，因为设计时几乎没有什么工作要做，唯一需要的外部支持部件是电容，以供芯片内部的变压器使用。这些变压器输出RS-232C传输所需的+12V和−12V电压，而不需要（额外的）外部电压调节器。图9-7所示是使用MAX3222的RS-232C接口的电路原理图。

电容C1必须大于或等于0.1μF。如果芯片在低于3.6V的电源下工作，C2、C3和C4也可以是0.1μF。而如果电源电压高达5.5V，则C2、C3和C4必须最小为0.47 μF，由于这些数字都是最小值，因此可以使用更大的电容。不过，如果C1增大了，则其他几个电容也必须相应增大。C5为VCC退耦，标准值是0.1μF。所有的电容都应该尽可能靠近芯片相应的引脚。

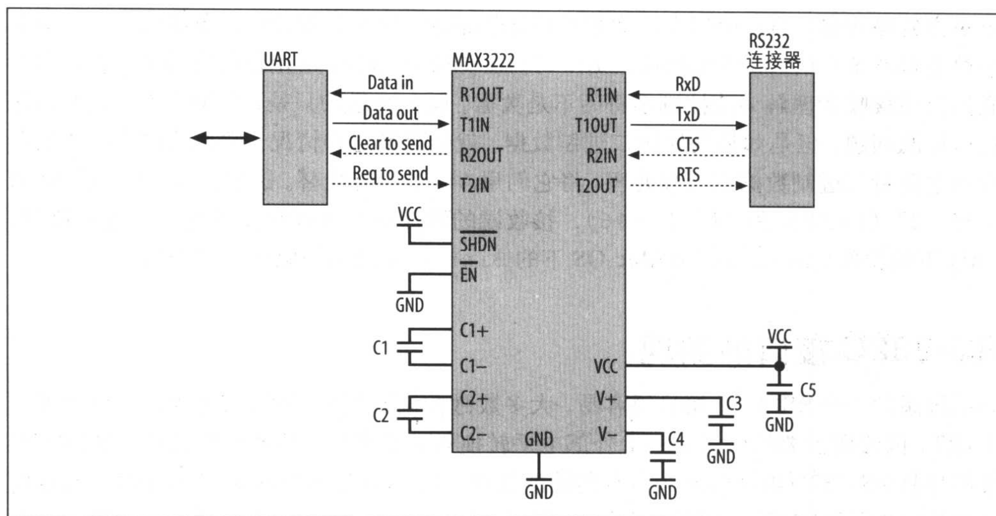


图 9-7：使用 MAX3222 的 RS-232C 接口

现在，剩下的连接就只有来自UART的串行数据线和到RS-232C连接器的信号线了。如果想实现一个最简单的串口，只需要 **Rx**、**Tx** 和地线就够了，**RTS**（发送请求）和 **CTS**（清除发送）是可选的。RS-232C 是一个 9 引脚或 25 引脚的连接器（它的外形看起来像字母D)，然而它也可以只是一排引脚、一个并行头，或者仅仅是直接焊接到印制电路板上的金属线。

MAX3222 有 2 个控制输入端：**$\overline{\text{SHDN}}$**（关闭）和 **$\overline{\text{EN}}$**（使能）。**$\overline{\text{SHDN}}$** 将 RS-232C 的发送器置为高阻抗，也就是将其禁止，这样就使芯片的电流消耗低于 1μA。当处于关闭模式时，接收器仍然处于活动状态，因此虽然此时 MAX3222 处于低功耗模式，UART 仍然能够接收数据。如果不需要使用 **$\overline{\text{SHDN}}$**，可直接将它连到 **VCC**。

与此类似，**$\overline{\text{EN}}$** 用来控制接收器的输出，**$\overline{\text{EN}}$** 置为高电平就可以使接收器的输出端进入高阻抗状态，同时发送器的输出不受影响。要想让接收器工作，只需将 **$\overline{\text{EN}}$** 激活（置低电平）即可。如果不需要将接收器置为无效，则可让 **$\overline{\text{EN}}$** 接地，也就是将其永远激活。

如果需要，可以通过一个微控制器的I/O线或采用锁存器的简单数字输出来控制 **$\overline{\text{SHDN}}$** 和 **$\overline{\text{EN}}$**。

MAX3222对于实现一个最简单的RS-232C接口来说已经足够了，它只需使用 **Rx**、**Tx** 和地线。另外，它额外还有驱动器支持 **RTS** 和 **CTS**，可以进行基本的流控制。如果你需要的是一个完整的 RS-232C 接口，MAX3241 是一个不错的选择。这个芯片的工作方式和

MAX3222相似，不过它的收发器还带有**DTR**、**DSR**、**DCD**和**RI**，可以对调制解调器进行控制。由于可以提供所要求的电压和电流，MAX3421还可以用来连接串行鼠标。

将串行端口用作电源

如果一个嵌入式系统要通过RS-232C串行接口一直和计算机主机相连，那么这个嵌入式系统可以顺便通过串口来供电。许多RS-232C信号端都没有使用，因此可以用来提供一定的电流，标准值是50mA。不过可能会因设备的不同发生变化，因此应该每次对所连接的特定系统进行检查。如果嵌入式系统要求的电流比这个值小，就可以RS-232C控制信号来提供电源。

例如，**RTS**（请求发送）或者**DTR**（数据终端请求）信号在许多RS-232C应用中都没有使用，可以用这些信号端中的一个作为电压调节器的电源输入端来为系统提供电压，这样计算机主机就可以使用串口的**RTS**来对嵌入式系统进行电源控制。在软件环境下，主机将**RTS**置为高电平，嵌入式系统就通电运行；将**RTS**置为低电平，嵌入式系统就断电关闭。所有这些操作都有一个前提，就是要确保嵌入式系统的输入电流足够小，小到可以通过**RTS**来供电。这种技术的好处是不需要外部电源来为嵌入式系统供电，只要接上串口就可以运行，好像是有不可思议的力量。另外还有一点要注意，就是这样的话你就不能将该RS-232C信号用作它本来的用途，这个引脚必须置为有效并一直保持有效，来为嵌入式系统供电。

图9-8所示为实现这种功能的电路原理图，图中也包括一个使用MAX3232来连接微控制器的RS-232C。注意二极管D1，由于**RTS**是低电平时将会是一个负电压（−15V），因此需要对电压调节器采取一些保护，因为调节器被设计为输入不能低于0V。D1可以是任何普通的整流二极管，比如1N4004，只有在**RTS**是正电压时它才会导通。电压调节器（MAX604）将来自**RTS**的电压转换为3.3V的电压，为嵌入式系统供电。如果需要的是5V的电源，则只需将电压调节器用MAX603代替即可，电路不变。电压调节器的输出通过C5进行平滑，通电即亮的LED可以告诉我们什么时候有电。MAX3232处在RS-232C端口和处理器之间，对来自处理器的串行传输和RS-232C进行逻辑电平的相互转换。

前面介绍了RS-232C的基本知识，它是一种很常见的接口，容易使用，但是仍有局限和不足之处。RS-232C最开始是用来将哑终端（dumb terminals）或电传打字机与调制解调器相连接的，而不是用于连接计算机和外设，因此连接计算机和外设更好的选择是RS-422。这个芯片被设计用来在对速度、功能和适应性要求更高的串行连接中使用。

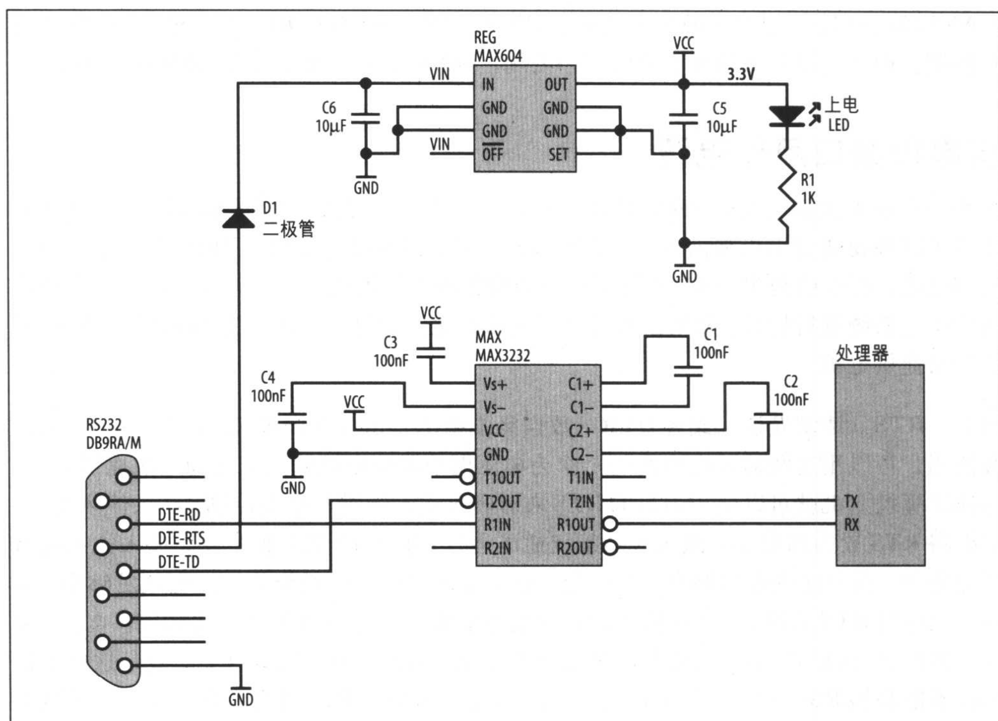


图 9-8：在低耗电嵌入式系统中用 RTS 供电

RS-422

RS-232C 的电压电平是相对于本地的地线而言的，而 RS-422 不同，它使用两根线之间的电压差来代表逻辑电平，这两根线称作双绞线或差分对（differential pair）。因此 RS-422 是一种平衡传输（balanced transmission），或者换句话说，它并不是相对于本地的地，任何噪声或干扰都将会同时影响两根双绞线中的每一根，但对二者之间差异的影响却很小，这种现象称为共模抑制。由于这个原因，RS-422 可以在更远的距离上以更快的速度传输数据，而且抗干扰能力却比RS-232C更强，RS-422传输数据的距离可达1200米。

图 9-9 所示是 RS-422 的基本连接示意图，驱动器（D）通过双绞线和接收器（R）连接。位于双绞线接收端的电阻Rt是一个终止电阻，它用来消除信号反射，信号反射发生在远距离传输过程中，并且是进行远距离传输所必需的。Rt 的标准值是 100~120Ω。

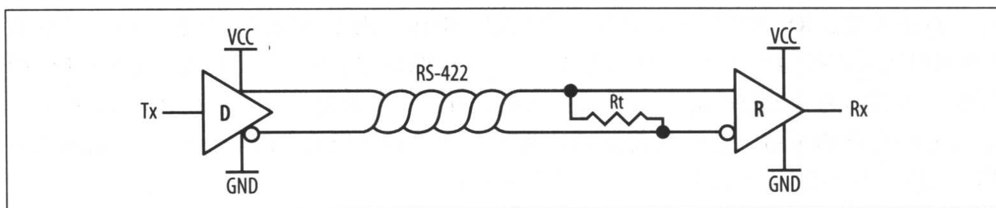


图9-9：RS-422

RS-422双绞线传输线之间的电压差在4V~12V之间（如图9-10所示）。RS-422在某种程度上和RS-232C是兼容的，通过将双绞线中负电压的一端接地，它就变成了非平衡传输，与RS-232C一样。由于RS-422的电压电平在RS-232C可接受的范围之内，所以这两种标准间有着一定的内部联系。RS-422是苹果公司的Macintosh计算机上的串口，但是随着该公司iMacs计算机的出现，这种串口逐渐变少了。

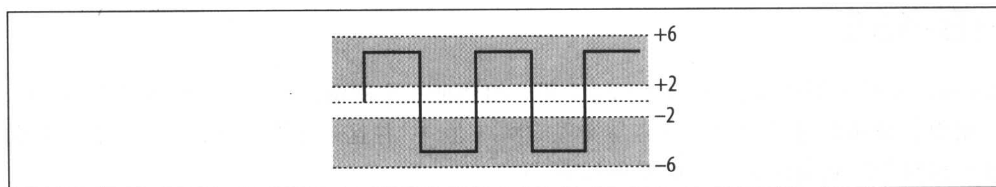


图9-10：RS-422的电压电平

可以使用的RS-422接口芯片种类繁多，图9-11所示是一种简单的RS-422双向接口，用两个Maxim公司的MAX3488来实现。每个MAX3488的**Tx**和**Rx**与嵌入式系统中的UART相连，就像使用RS-232C时那样。

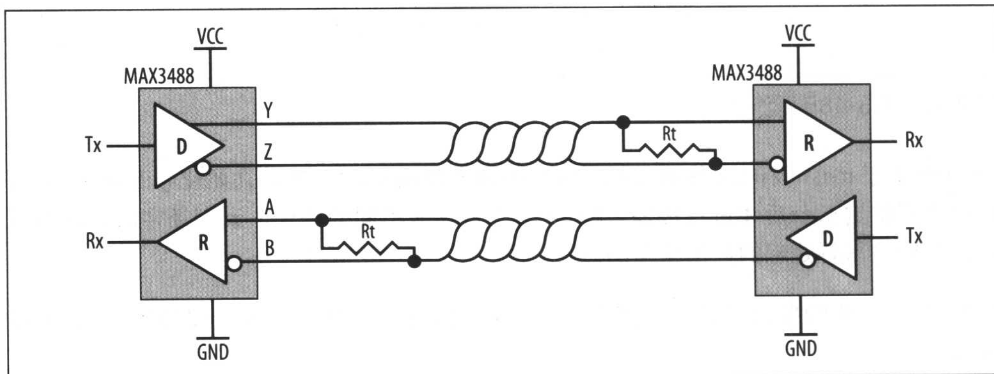


图9-11：双向RS-422接口

有一点很重要：RS-422只是为这种标准规定了电压，而不是物理连接和具体实现细节（插脚引线或连接器），后面这些内容由RS-449标准规定。现在，没有人会为RS-449而烦恼，主要因为这种标准对大多数用户来说具有不必要的复杂性。使用RS-422的人只是做他们自己的事情，选择任何他们认为适合自己的应用的电缆和连接器（和插线引脚）。RS-422好像已经成为了自我表现的代名词。

有些RS-422接口芯片有一个可选择的使能输入端，当这个引脚有效时，芯片产生输出并驱动数据发送到双绞线上。当它无效时，芯片的输出端就变为高阻抗，整个芯片就成为“不可见”的，如同不存在一样。由于芯片可以从连接中“消失”，因此就有多个接口芯片（同时也就有超过两个的嵌入式系统）可以和双绞线相连，这样一来，RS-422就可以扩展成为一个低成本、功能强大的简单网络。当RS-422用这种方式实现时，它就变成了RS-485。

RS-485

RS-485由RS-422演变而来，RS-422常用于组建低成本网络，并且普遍被用在许多工业应用中。RS-485是最简单和最容易实现的网络之一，并且还允许多个系统（节点）仅通过一根双绞线来交换数据，如图9-12所示。

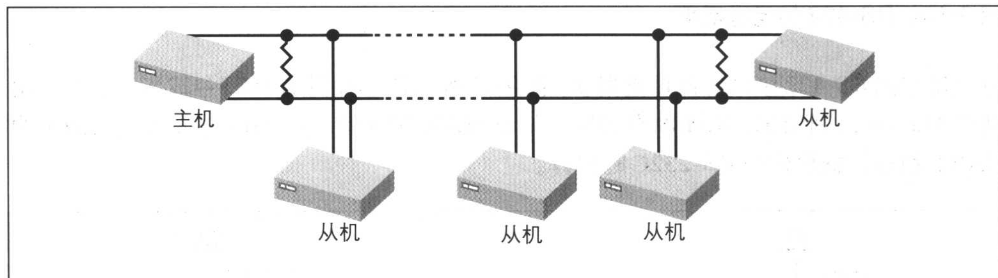


图9-12：RS-485网络

RS-485基于master/slave体系结构。所有的事务都由master进行初始化，而slave只有在接收到特定指令后才进行传输。RS-485支持许多不同的协议，人们经常做自己的事情，并制定与将处理的应用相关的协议。

到RS-485网络的接口通过一个收发器来实现，诸如一个Maxim MAX3483，如图9-13所示。

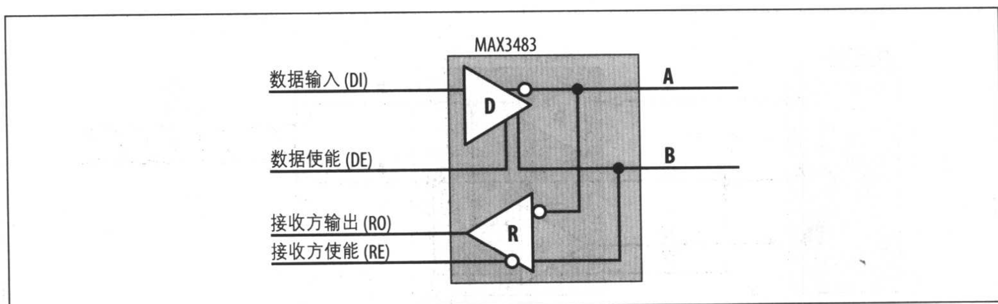


图 9-13：RS-485 收发器

MAX3483只是一个能够输入的RS-422收发器，在设计中使用起来很简单。在网络一端，MAX3483有两个信号线**A**和**B**，它们是双绞线（网络电缆）的接入点。MAX3483还有**DI**（Data In，数据输入）和**RO**（Receiver Out，接收方输出）两个引脚，分别与UART（或微控制器）的**Tx**和**Rx**信号相连接。

由于是连接到一个公共网络，它必须能够在这个网络上进行监听和传输，所以收发器有两个控制输入端：**DE**（Data Enable，数据使能）和**$\overline{\text{RE}}$**（Receiver Enable，接收方使能）。对**DE**输入一个高电平，则允许**DI**输入端的数据在网络上传输；对**DE**输入一个低电平，则禁止**DI**的输出在网络上传播。同样地，对**$\overline{\text{RE}}$**输入一个低电平，就使接收器可以接收数据，这样网络上的数据都流向**RO**。通常情况下，**DE**和**$\overline{\text{RE}}$**由处理器的一个I/O线控制着。现在你已注意到，**DE**是高电平有效，而**$\overline{\text{RE}}$**是低电平有效。这种情况并不是偶然的。处于网络中的一个节点不会在发送数据时接收数据，反过来，也不会在接收数据时发送数据。因此，在任何一个时刻，一个节点要么处于发送状态，要么处于接收状态。如果正在发送数据，就不会接收，反之亦然。也就是说，对发送的控制和对接收的控制在逻辑上是相反的。通过对**DE**置高电平而对**$\overline{\text{RE}}$**置低电平，只通过一根控制线就可以控制接收和发送。图9-14所示为一个MAX3483通过这种方式和一个微控制器进行信息交换的过程。微控制器通常置**DE/$\overline{\text{RE}}$**为低电平来监听网络流量。若它希望发送数据时，就置**DE/$\overline{\text{RE}}$**为高电平。当数据发送结束，就又置**DE/$\overline{\text{RE}}$**为低电平，重新对网络进行监听。

RS-485可以实现为半双工，其中单根双绞线既用来发送数据，又用来接收数据（如图9-15所示）。也可以作为全双工来实现，这时发送和接收数据由不同的双绞线来完成（如图9-16所示）。全双工的RS-485有时候就是所谓的四线模式（four-wire mode）。注意，对于全双工操作，MAX3483被MAX3491代替，后者带有双网络接口。

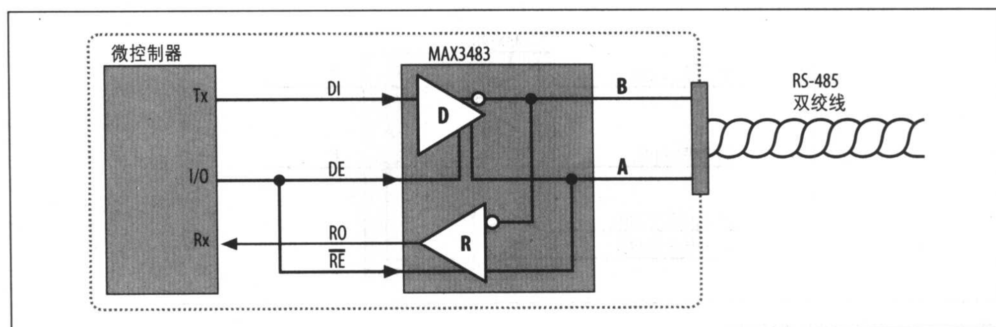


图 9-14：连接 MAX3483 和微控制器

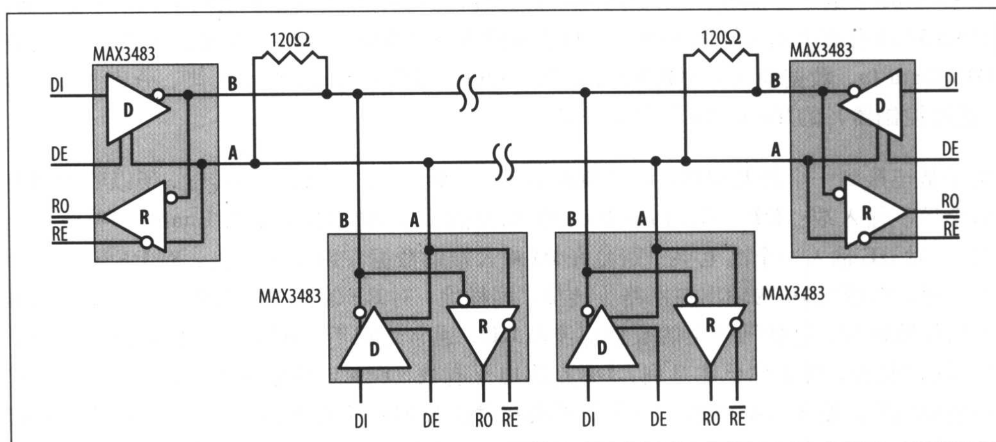


图 9-15：半双工 RS-485

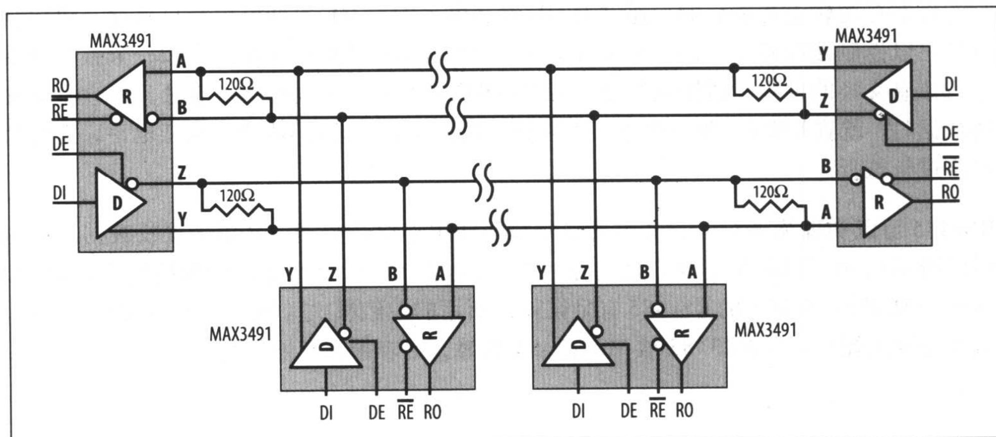


图 9-16：全双工 RS-485

这些示例显示了四台计算机（节点）连接到一个 RS-485 网络的情况。每个 RS-485 接口芯片（MAX3483 或者 MAX3491）都位于一个单独的嵌入式计算机中。在每个嵌入式系统中，UART 的发送方输出端 Tx 连接到每个 RS-485 接口芯片相应的 **DI**（Driver In）。与此类似，**RO** 连接到每个 UART 的输入端 **Rx**。每个 RS-485 接口芯片的驱动器的使能是通过对 **DE** 的设置实现的，同样，接收使能是通过对 $\overline{\mathbf{RE}}$ 的设置来实现的。

通常情况下，所有连接到 RS-485 网络的系统都将它们的接收器置于活动状态来监听网络。只有当一个系统想发送数据时，才将其驱动器激活。许多正式的协议都将 RS-485 作为一个传输媒介来使用，它在正式协议中的使用率是在一些个人制定的协议中的两倍。你需要避免的主要问题就是网络中两个节点可能同时发送数据。解决这个问题最简单的方法就是指派一个节点为主节点，而其他所有的节点均为从节点。在一个网络中，只有主节点可以进行数据的发送，而一个从节点只能在主节点的数据发送完毕之后对主节点进行响应。

网络上可能拥有的节点数目受制于接口芯片的驱动能力。通常情况下，这种限制是每个网络 32 个节点，不过有些芯片可以支持的节点数目可达 512 个。

第10章

IrDA

弃暗投明和弃明投暗是
截然不同的。
——大卫·林赛
《大角星之旅》

上一章已经介绍了通过铜线来实现串行通信，在本章里，我们来看一下使用红外线来进行短距离的数据传输，而无需通过电缆连接的串口。数据的红外（Infrared，IR）传输越来越常见，在手提电脑、PDA和移动电话中都可以看到IR收发器。在一些外设，如打印机和网络接口中也有IR收发器，可以让人们在移动时进行不需要电缆的连接。IR通信还用于通过远程控制与各种电器进行对话。你的电视机、录像机和DVD的遥控器都有一个红外LED，来向这些电器发出指令。

我们从最通用的标准开始对IR通信进行讨论，稍后还会了解简单的红外硬件是如何实现的。

IrDA简介

IrDA是常用于计算机和外设中的红外传输标准。IrDA代表红外数据协会（Infrared Data Association），这个联盟由150多个公司组成，制定并发展了IrDA标准。IrDA标准起源于Hewlett-Packard计算器中使用的红外通信连接，这个连接又称为HP-SIR（Hewlett-Packard Serial Infra Red，Hewlett-Packard串行红外）。在HP-SIR的基础上，IrDA标准进行了很大的扩展，可以提供一系列的协议供应用软件在通信时使用。

制定IrDA标准的初衷是为了在设备与设备之间提供近距离的通信。移动设备，例如手提电脑，当必须要和其他机器或网络连接时就面临一个问题：电缆碰巧不在手边，或者一个机器没有配置好，不能进行网络连接。当使用者是非技术人员时，这种情况就成为

一个实实在在的问题，而IrDA标准就是为解决这个问题而开发的。利用IrDA协议，不需要电缆，标准协议确保设备不连接就可以交换信息。关于IrDA标准和协议的详细信息可以从 *http://www.irda.org* 网站得到。

开发IrDA标准是为了让使用者成为可移动的工作人员，使用手提电脑或者PDA和附近的其他计算机、PDA或外设进行通信。这个概念带来了许多重要的结果。进行通信的设备在物理上是接近的，因此所需要的仅仅是低功耗传输。这点很重要，因为红外辐射所能达到的最大强度受相关的物理规则控制，同时，协议也合理地假设要通信的两个设备在物理上相互指向对方，且有优先使用权（如果猫不具有特殊的反射能力，你将遥控器指向它不会改变电视的频道）。这个协议还假设只有两个设备可以同时进行通信，并且由于它们位置上非常接近，可以不用考虑其他IrDA设备的干扰。因此，IrDA协议没有必要处理传输中的冲突和检测其他标准，比如802.11（无线以太网）中必须处理的问题。可以有两个IrDA设备在桌子的一端，而另两个IrDA设备在桌子另一端进行通信，不会有任何问题。进一步说，一个传输由使用者发起，这样就简化了软件协议。一个总的指导原则是：IrDA 实现起来成本应该比较低，因为它必须在低功耗设备中找到自己的出路。

请记住，IrDA就是一个点对点的协议，用在近距离进行异步串行传输。最初的IrDA规范（1.0）支持的数据传输率在2400bps~115.2kbps之间，传输的距离为1米（尽管有些IrDA收发器的传输距离超过1米）。红外通信最初的标准速率是9600bps，设备可以根据它们的能力和需要的速率进行协商，以更快或更慢的速率进行传输。与RS-232C不同，使用者无需进行设置、了解，甚至考虑正在进行的通信速率是多少。

IrDA发送器传输数据时发出的红外光每一边的角度在15°~30°之间，接收器光线每一边的“视角”为15°（如图10-1所示）。所以，如果两个IrDA设备相距1米远或者更近，而且相互对准对方，通信就不会有问题。从最初的规范开始，这个标准就已经被扩展到支持1.15Mbps~4Mbps这样更高的数据传输率。

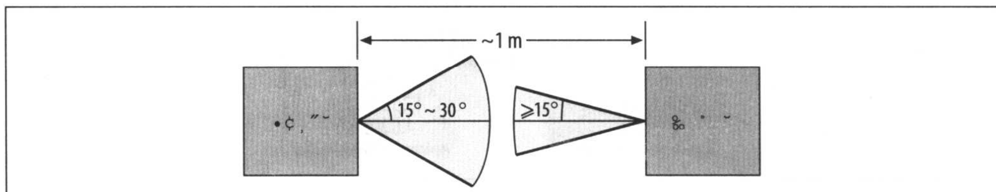


图 10-1：IrDA 传输和视角

IrDA标准为通信规定了许多协议层，*IrPHY*（红外物理层）规范对硬件层进行了详细规定，包括在传输之前对UART输出进行调整的要求。控制协议叫作高级数据链路控制规程（High-level Data Link Control，HDLC）。IrLAP（Infrared Link Access Protocol，

红外连接访问协议）用HDLC来控制对通信介质的访问。每个设备有一个IrLAP，IrLAP连接是主/从配置所最基本的，或者用IrDA的话说，它们是主设备和次设备。主设备发起一个传输，发送命令，处理数据流控（握手）。主设备不是计算机而是其他任何设备的情况很少见，次设备（比如打印机）只是对来自主设备的请求进行响应。两个主设备通信时，其中一个会充当次设备的角色。一般情况下，发起传输的设备保持主设备角色，另外一个在通信期间变成次设备。

根据IrLMP（Infrared Link Managment Protocol，红外链路管理协议）可以开发一个设备软件，这个软件为希望使用IrDA进行通信的多个任务提供了共享同一个IrLAP的方法。IrLMP还提供了一个询问协议，使用这个协议，一个设备可以对另一个设备进行询问，以确定远处系统可以提供哪些服务。这个询问协议叫作LM-IAS（Link Management Information Access Service，链路管理信息访问服务）。前面说的这些都是基本的IrDA协议，所有设备都必须支持。除了它们，IrDA还提供了许多可选的服务。IrCOMM提供对标准串行端口和并行端口设备的模拟，在应用软件中，IR端口可被当作另外一个串行或并行端口来使用。使用IrCOMM，手提电脑或PDA可以与一台带有红外功能的打印机通信，就好像这台打印机和移动计算机在物理上是相连的一样。IrLAN允许通过红外接口对局域网进行访问，在支持面向对象编程的软件中，IrOBEX提供了一种在设备之间进行对象交换的机制。最后要说的是TinyTP，这是一种小型协议，它可以在数据从一个设备发送到另一个设备时进行流控（握手）。图10-2给出了这些协议层是如何集成到一起的。

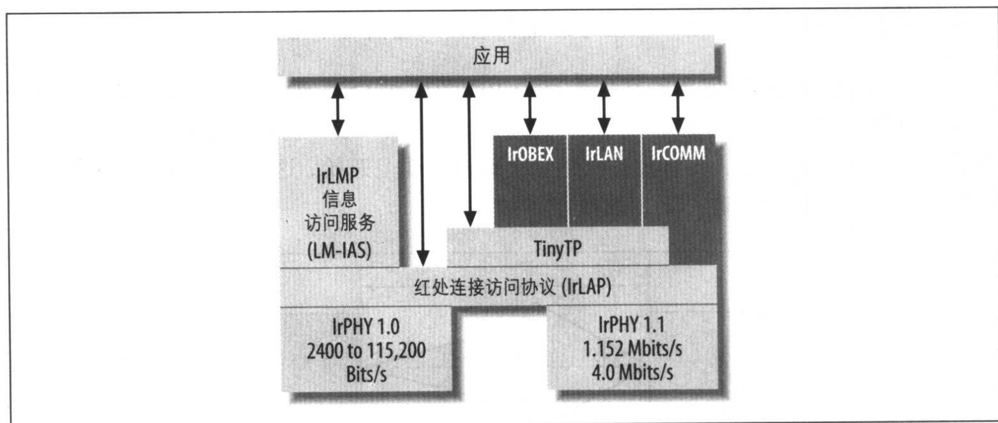


图10-2：IrDA协议层

在比较低的数据传输率上，所有的协议处理、数据打包、错误检测都是通过支持IrDA的设备的处理器在软件下完成的。在较高的传输速率上，这些功能由专门的硬件来实现，因为低成本的嵌入式处理器速度有限，无法在一定的时间内完成这些功能。

由于IrDA通信是通过光来实现的，因此在传输中必须有某种识别逻辑0和1的方法。为了解决这个问题，IrDA使用一种叫作归零（Return-to-Zero，RZ）的位编码机制。根据RZ的规定，一帧是一个被分为多个子时间片的时间片，每个子时间片代表一个单独的比特。逻辑0由一个3/16子时间片宽度的脉冲表示，而无脉冲则表示逻辑1（如图10-3所示）。

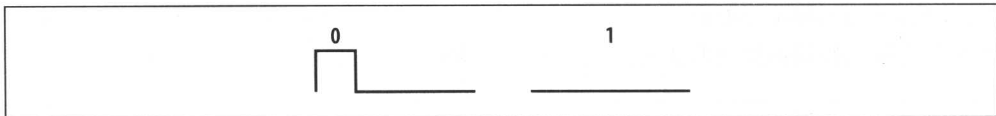


图10-3：RZ编码

在数据传输率为4Mbps时，脉冲相位调制（Pulse Position Modulation，PPM）被用来对不同的位进行识别。根据PPM，脉冲的相位可以发生改变，它在子区间中的位置决定了传输的位模式。在IrDA中使用的PPM称为4PPM，这是由于它使用其4个位置中的一个来发送两个数据位（如图10-4所示），在PPM术语里面，这叫作一个单元。

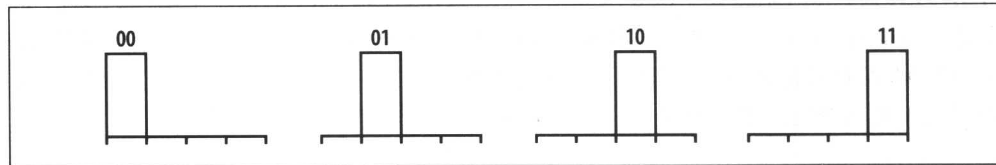


图10-4：4PPM单元编码

图10-5所示是一个数据包的实例（对于4Mbps传输）。它由一个64单元（128位）的引导包、一个开始包、要传输的数据组成的数据帧体、一个32位的循环冗余校验（CRC）码和最后的包停止标志组成，其中数据帧最小为2字节，最大可以为2050字节。

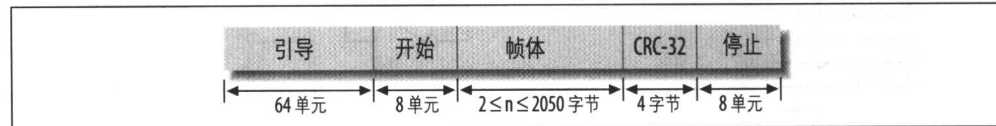


图10-5：一个4Mbps的数据包

目前，大多数UART都无法用RZ或PPM编码进行传输。因此，一个叫作EnDec（EncoderDecoder，编码－解码器）的特殊设备可以将标准的UART输出和RZ脉冲进行相互转换。Agilent公司的HSDL-7001和Microchip公司的MCP2120都是很不错的EnDec。有些UART，比如MAX3100，在芯片中内置了一个EnDec，因此可以直接和红外收发器进行连接。

IrDA 接口

对于红外发送和接收，可以使用一个单独的红外LED和红外光敏二极管探测器，也可以使用集成了红外LED和光敏二极管的组合红外收发器，连同支持部件一起打成一个方便使用的包(如图10-6所示)。有几个生产商 —— 包括Agilent公司(*http://www.agilent.com*)和Vishay(*http://www.vishay.com*) —— 生产这类设备。作为收发器包的一部分，接收红外光的光敏二极管被黑色的滤光器遮住，用来滤除可见光，提高红外光的接收能力。

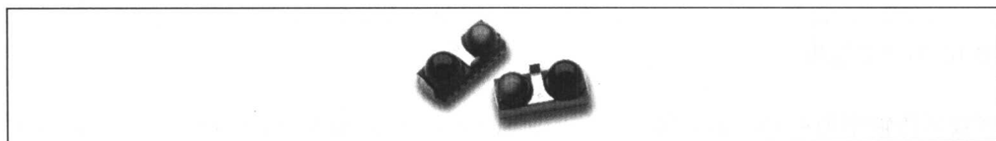


图10-6：Agilent IrDA 收发器

MAX3100是个通用UART，可以使用它向嵌入式计算机添加RS-232C或RS-485接口。MAX3100通过SPI总线和主机处理器相连，可以在2.7V~5.5V的电压下工作。不过，这个芯片还支持IrDA，经过配置可以输出RZ编码的数据，也可以接收RZ编码的比特流。实现IrDA接口时需要做的只是添加一个红外收发器、一些反相器和几个支持部件。图10-7所示是这个电路的原理图。

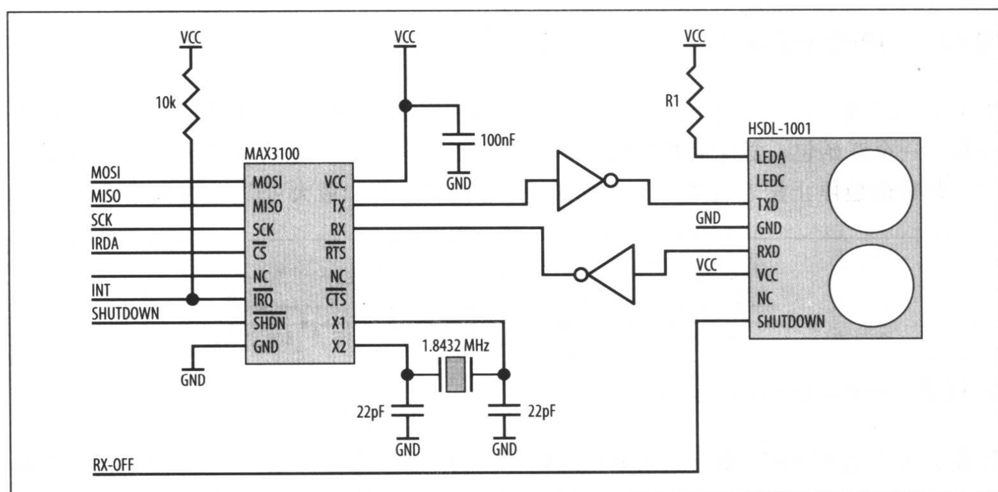


图10-7：嵌入式计算机的IrDA接口

图10-7的左边是到微控制器的标准SPI连接，MAX3100有一个中断输出端，通过这个输出可以通知主机处理器状态的变化(比如接收到了数据)，一个10kΩ的电阻将这根中断信号线的电压置为高电平。MAX3100还有一个关闭输入端，可将该设备置为低耗电

模式，这个信号可以通过微控制器的一根I/O线驱动。MAX3100需要一个晶体来产生发送和接收时的时钟信号，振荡频率可以是1.8432MHz或3.6864MHz，通过软件对内部时钟分频器的控制，这两种频率都可以用来产生任何需要的波特率。低频的晶体会让MAX3100耗电少一些。

IR收发器的种类有很多，在图10-7中所使用的是Agilent公司的HSDL-1001。想要连接MAX3100和HSDL-1001，只需将发送（**TX**）和接收（**RX**）信号同时反相即可。HSDL-1001有一个关闭输入端，用来将接收的光敏二极管置于低耗电模式，而对发送的LED没有任何影响。这个关闭输入端也可以由处理器的一根I/O线来驱动。为了使功能尽量多样化，这个关闭输入端的控制独立于MAX3100的关闭控制。发送端HSDL-1001的LED需要串联一个限流电阻R1，这个内部LED电路和第4章中的LED电路基本上差不多。当LED发光的时候，其阴极电压最低为2.1V，而最大电流是240mA，因此根据欧姆定律，电阻R1（当工作在5V的电源下时）大约为15Ω。

最后要说的一点是，IrDA对噪声和干扰都很敏感，因此对每个电源引脚都应该用电容进行适当的退耦，还需要用地线来屏蔽收发器和相关的信号。

其他红外设备

电视机、录像机、DVD、空调和许多其他设备都有接收来自遥控器的命令的红外端口，但不好的一点是它们中没有一个（或者说只有极为少数几个）和IrDA是兼容的。电器制造商们都按他们自己的意愿来做，还经常采用一些他们自己规定的非通用的波特率，因此前面说的与IrDA兼容的电路在一种特定的电器中可能会正常工作，也可能无法工作。不过，如图10-8中那样简单的电路却可以让你不必再担心这个问题。

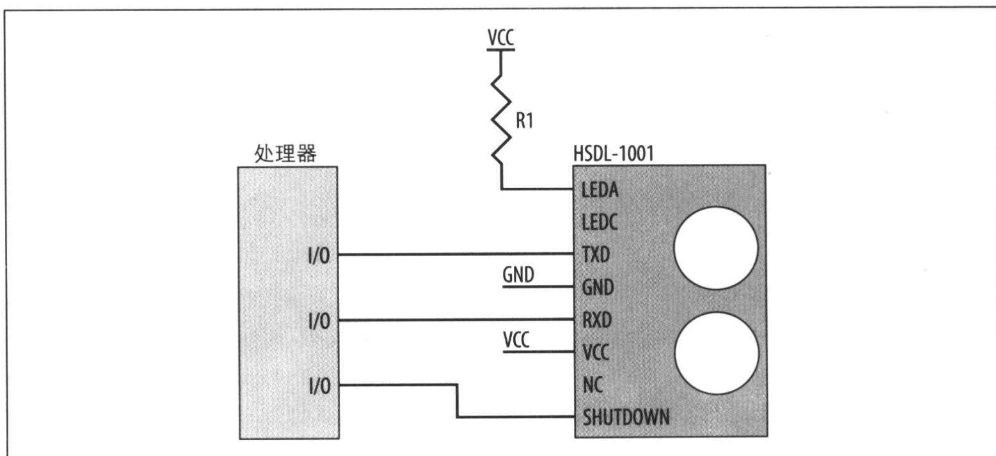


图10-8：红外接口略图

HDSL-1001的发送器可以用一根处理器的I/O线直接驱动，同样，其接收器也可以用一根I/O线（作为输入）来采样。因此，在软件控制下（适当的时候需要手工切换发送器I/O线），HSDL-1001可以在适当的比特率上接收RZ比特流。这种手动技术通常用在标准串行接口中，用来在没有UART（并且也不能通过扩展来添加）的处理器上实现一个串行端口。这种软件UART技术其实可以很容易地扩展成一个红外接口，这就取决于作为程序员的你能否确保传输时的时序正确。

注意： 裸眼无法看到遥控器发出的红外光，但如果拿摄像机对准它就可以看到了。将遥控器对着摄像机的镜头按下一两个键，做这些的时候看着探视镜，你就可以清楚地看见代表控制比特的光线。出现这种现象的原因是摄像机里面的CCD（电荷耦合器件）图像对红外光像对可见光一样敏感，这也是摄像机即使在光线比较暗的情况下也可以拍摄得比较理想的原因。为进一步提高它们在夜间对图像的拍摄能力，有些摄像机的前端还有一个红外光源，用来将黑暗“照亮”，而这些对于人来说都是看不到的。

用遥控器做一下这个试验，你会很惊奇地发现红外LED的光线实际是那么的亮。

关于电器（和遥控器）IR协议及编程方面的更多信息，可以到*http://www.remotecentral.com*中查找。

第11章 USB

万物生于有，万物皆有因。

——留基伯

《原子论》

在第9章里，我们了解了RS-232C这个历史悠久但不够权威的通信标准。RS-232C存在许多问题和限制，让两个RS-232C设备进行通信并不是一件容易的事情。你需要用正确的电缆，而且连接器的类型也要正确，然后还需要手动调节通信参数，比如数据传输速率、奇偶校验以及握手方式等。它充其量只能算是个令人讨厌的东西，最糟糕的情况下还是一件令人头疼的事情。而对于硬件制造商来说，他们也是处于进退维谷的境地。在开发产品的时候，目标是让这些产品尽可能地容易使用，你不会希望使用者拿着错误的电缆犹豫不定，进行手工设置失败而没能将你的产品很好地和其系统结合起来，这些情况会令他们很不愉快。

通用串行总线（Universal Serial Bus，USB）是解决这些问题的办法。USB让外设和计算机使用标准协议以标准方式进行连接，并且使外设的即插即用成为可能。USB很快占领了台式机市场，将RS-232C逼到了危险的境地。苹果公司的Macintoshes计算机已经不再有RS-232C/RS-422端口，不久所有的PC机都会这样做。因此，如果想把你的嵌入式计算机和未来的台式机连接起来，了解USB（并知道如何构建一个USB端口）是一件很迫切的事情。USB支持对打印机、调制解调器、鼠标、键盘、游戏杆、扫描器、数码相机和其他许多设备的连接。

USB使许多事情变为可能，不过对它的开发要比RS-232C复杂一些。USB具有不需要进行手工设置设备就可以与计算机主机的操作系统相结合的好处（对使用者来说）。然而，它对软件来说并没有多增加一层复杂性，因为嵌入式的软件代码必须通过适当的方

式和主机交互。USB甚至可以通过发送数据的同一根电缆为外设供电，而不需要外部电源（或电源线），因此，对使用者来说，一个USB设备就是简单的代名词。

在这一章里，我们首先对USB进行概述，然后接着介绍如何在嵌入式系统中内置一个USB接口。USB协议和规范的内容很多很复杂，已经超出了本书的范围。值得庆幸的是，设计基于USB的硬件是一件简单得多的事情，我们将只是大概了解一下，然后看看一个物理USB的实现过程。要想全面了解这个标准、获取产品的生产商信息以及更多的相关文档，请访问站点 *http://www.usb.org*。

USB 简介

USB规范有两种：USB 1.1和USB 2.0。USB 2.0与USB 1.1完全兼容。USB 1.1支持的数据传输率为12Mbps~1.5Mbps（用于慢速外设），USB 2.0支持的数据传输率可达480Mbps。USB传输可以同步（注1），也可以异步进行。

USB是一种高速总线，它连接的设备数量最多可达127个（如图11-1所示）。计算机上只能有一个或两个端口的限制已经不再存在，而且RS-232C关于电缆的使用很混乱，但USB关于电缆和连接器的标准杜绝了这类问题。设备能够自动向计算机主机表明身份，而且可以热切换（hot-swap），也就是说不需要系统断电就可以和设备进行连接或断开。

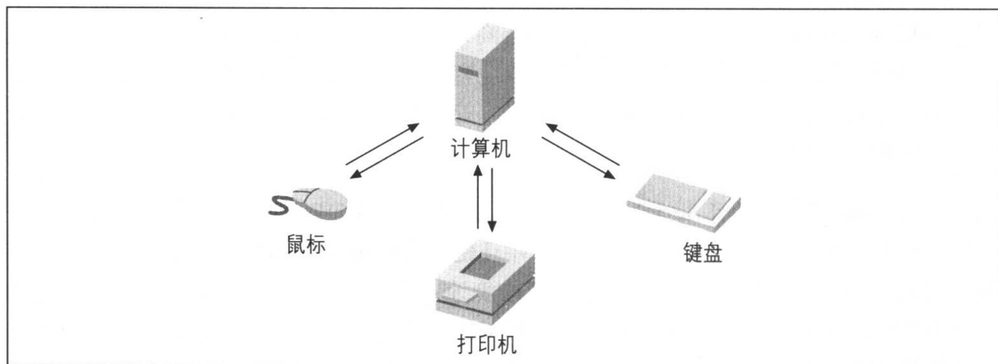


图 11-1：USB 可以让主机连接多种外设

USB网络的基本拓扑结构是星型的，一个USB系统由一个或多个USB设备（外设）、一个或多个集线器（hub）和一个主机（控制计算机）组成。计算机主机有时又叫作主控

注 1： 意味着以相等的时间间隔发生。

制器，在一个USB网络中只能有一个主机。主控制器内置了一个根集线器，根集线器提供主控制器上的初始附属点。这些集线器成为网络中的节点，这些节点与设备或其他集线器连接，在USB通信中它们基本上是透明的，换句话说，就是集线器的存在并不会影响设备和主机之间的通信。

集线器用来扩展USB网络。例如，一个特定的计算机主机可以有5个USB端口，通过在主机的端口上连接集线器（每个集线器有多个端口），系统的物理连接能力就提高了（如图11-2所示）。许多USB设备（如键盘）都内置有集线器，既可实现其本身的功能，又可进行扩展。

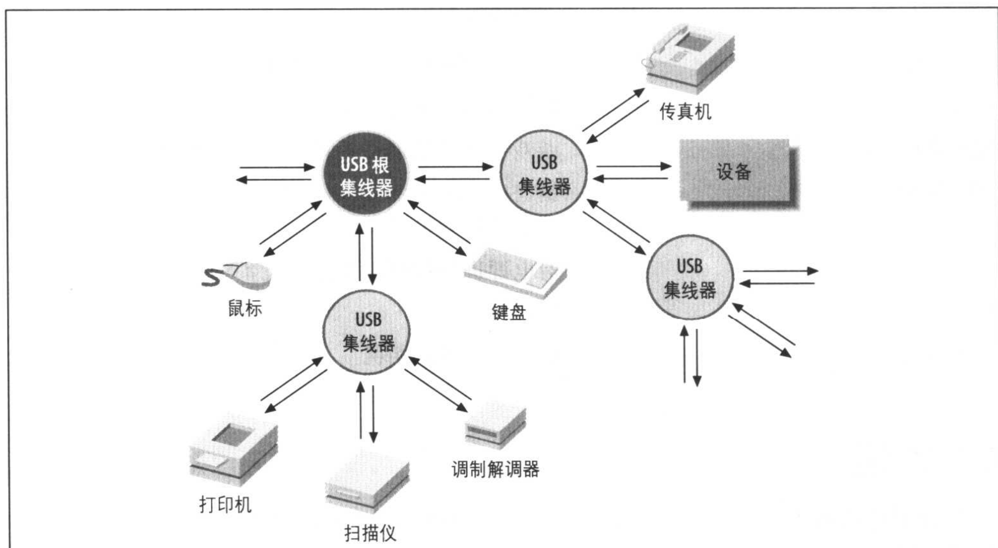


图11-2：使用集线器的可扩展USB

主机定时对集线器的状态进行查询，当一个设备新接入一个集线器时，这个集线器就会向主机报告状态改变，主机发出一个命令使该端口有效并对其进行重新设置。位于这个端口上的设备进行响应，主机收到关于设备的信息，根据这些信息，主机的操作系统确定对这个设备使用哪种驱动程序，接着设备被分配一个唯一标识的地址，主机向它发出内部设置请求。当一个设备从总线上移走时，主机就从其可用资源列表中将这个设备删除，主机对USB设备的探测和识别叫作总线列举（bus enumeration）。

USB可以“感知”和支持不同类别的设备，每种类别代表这种设备能够为主机提供的功能。表11-1列举了一些类别。一个USB设备可以同时属于几种类别。

表 11-1：USB 设备类别

类别	用途
音频	音频和音乐设备、声音系统
芯片 / 智能卡接口设备（CCID）	智能卡设备
通用类	一般设备
通信设备	调制解调器、电话和网络接口
HID	人机接口设备（HID），比如鼠标和键盘
集线器	USB 集线器
IrDA	红外设备
海量存储	硬盘、CD-ROM 和 DVD-ROM
监视器	计算机监视器和显示设备
物理接口设备	游戏杆和其他提供物理反馈的设备（如运动平台）
POS 终端	销售点（Point-of-Sale，POS）设备，如收银机和资金电子过户设备
电源	对电源进行控制或监控的设备(如后备电池和充电)
打印机类	打印机
图像类	扫描仪和数码相机

USB 包

USB 数据传输有四种类型：控制传输对总线和总线上的设备进行设置，并返回状态信息；块传输通过USB设备对数据进行异步发送；同步传输用来发送对实时性要求较高的数据，如送往一个输出设备的音频数据，与块传输（这种传输是双向的）不同，同步传输是单向的，并且不包含循环冗余校验（CRC）字段；中断传输用来在一定的时间片发送数据，这个时间片的范围为 1~255 毫秒。

数据在USB 设备之间以包的方式进行传输，一个传输由一个或多个包组成，一个包内包括 1 个 SYNC（同步）字节、1 个 PID（包 ID）、内容（数据、地址等）和 1 个 CRC 字段。

SYNC 字节锁定接收器时钟的相位，它相当于 RS-232C 帧的起始位。PID 指出包的功能，例如它是否是一个数据包或配置包。包 ID的高 4 位是低 4 位取反的结果，这样做是为了增加一种错误检测机制，比如一个数据包的包ID会是`0x3C`，也就是二进制的`%0011 1100`。

USB 包可以是以下四种类型之一：令牌、数据、握手或先导包。

令牌是个24字节的包，它决定总线上发生的数据传输的类型，有四种类型的令牌包（如图11-3所示）。一个令牌包由1个SYNC字节、1个包ID（表明包的类型）、主机访问的设备的地址、终端地址和1个5位CRC字段。终端地址是设备中数据的内部地址。

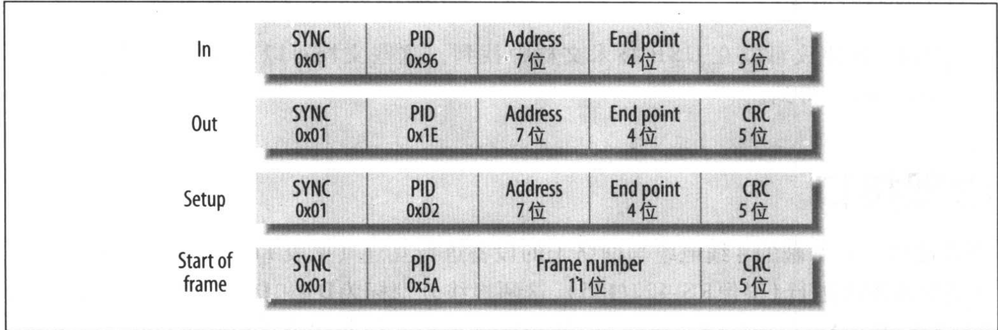


图11-3：USB令牌包

数据包有两种类型，分别为DATA0和DATA1（如图11-4所示）。在传输过程中这两种包相互交替，一个数据包可以传输的数据为0~1023位。数据包的CRC是16位。

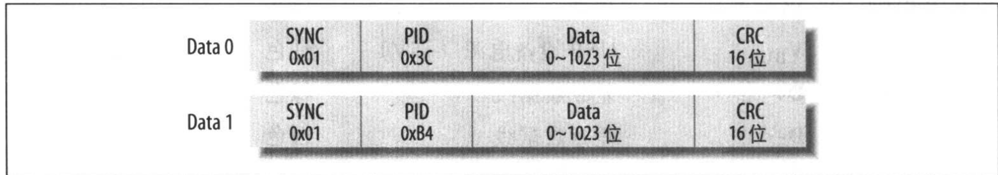


图11-4：USB数据包

握手包有三种类型（如图11-5所示），成功接收到数据后通过应答（Ack）包进行确认，当数据接收失败时，接收端通过向主机发送否认（No Acknowledge）包来通知对方。

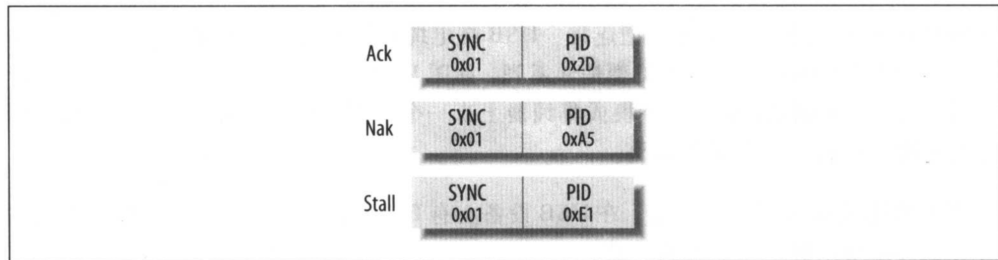


图11-5：USB握手包

描述符是一个用来将设备的性能通知主机的数据包，它包含1个设备生产商的标识符、1个产品标识、1个类别类型和设备的内部设置，比如终端和对电源的需求。每个生产商都有一个唯一的ID，每个产品依次也有一个唯一的ID。计算机主机上的软件根据从描述符中得到的信息判断一个设备有哪些功能，以及主机如何与设备进行交互。

详细的USB协议可以在USB技术文档中找到，这些文档可以从USB站点（*http://www.usb.org*）得到。

物理接口

USB使用一根屏蔽的4线电缆与网络上的设备进行互连（见表11-2）。数据传输通过一根差分双绞线进行（很像RS-422/485），这两根线分别标为**D+**和**D-**，另外两根线是V_{BUS}和**GND**，其中V_{BUS}向USB设备供电。使用USB电源的设备称为总线供电（bus-powered）设备，而使用自己外部电源的设备叫作自供电（self-powered）设备。为了避免混淆，USB电缆中的线都用不同的颜色作标记。

表11-2：USB线

连接器引脚	信号名称	用途	颜色
1	V_{BUS}	USB设备电源（+5V）	红色
2	**D+**	差分数据线	绿色
3	**D-**	差分数据线	白色
4	**GND**	电源和信号地	黑色

有些USB芯片将**D+**和**D-**分别表示为**DP**和**DM**。

从一个设备连回到主机称为上行（upstream）连接，同理，从主机到设备的连接叫作下行（downstream）连接。为了防止回环情况的发生，上行和下行端口使用不同的连接器。连接USB网络的唯一方法是星型连接。USB在电缆和设备的连接中使用两种类型的插头和两种类型的插座，第一种类型是A系列，如图11-6所示。A系列的连接器用来进行上行连接，换句话说，就是在主机或集线器上有一个A系列插座，而在连到主机或集线器的电缆一端有一个A系列插头。

B系列的连接器如图11-7所示。在USB设备上有B系列插座，而B系列的插头在从主机或集线器接出的下行电缆的一端。

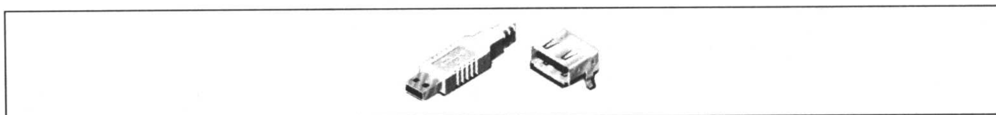


图 11-6：A 系列插头和插座

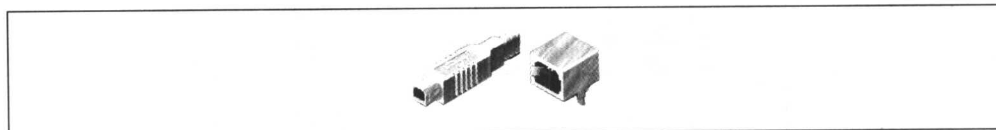


图 11-7：B 系列插头和插座

图 11-8 给出了两种连接器的实际连接方式。这样可以确保 USB 设备、主机 / 集线器和 USB 电缆始终以正确的方式连接，而不可能出现电缆接入方式出错，或直接将两个 USB 设备连接到一起的情况。

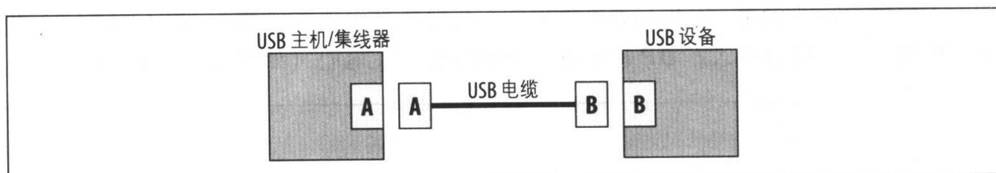


图 11-8：USB 连接器和电缆

由于集线器以下行方式连接 USB 设备，同时又以上行方式连接 USB 主机或集线器，因此两种类型的插座它都有（如图 11-9 所示）。

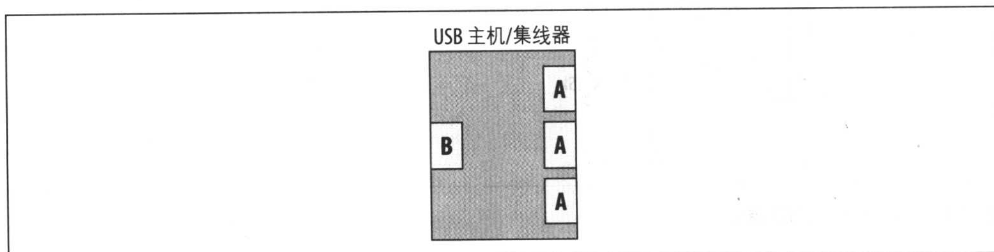


图 11-9：USB 集线器上的插座

实现 USB 连接的芯片只需要很少的外部部件作为 USB 端口，图 11-10 所示为一个上行端口的示意图。

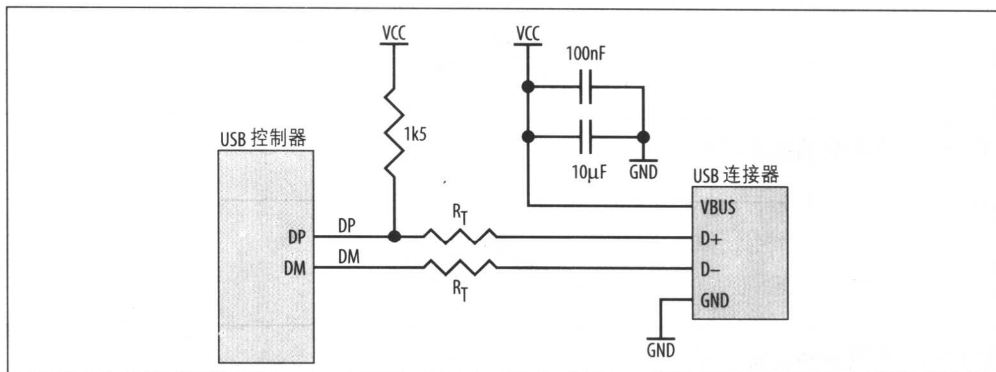


图 11-10：上行 USB 端口

在这个例子中，嵌入式系统通过 USB 端口来供电。如果嵌入式计算机有它自己的电源，则不进行 **VCC** 与 USB 连接器的引脚 1（$\mathbf{V_{BUS}}$）的连接。连接到 **DP** 的电压上拉电阻只有在上行端口中才需要，因此如果你现在要在集线器上实现下行端口，就不需要这个电阻。但是，下行端口要求在 **DP** 和 **DM** 上都必须接下拉电阻（如图 11-11 所示）。

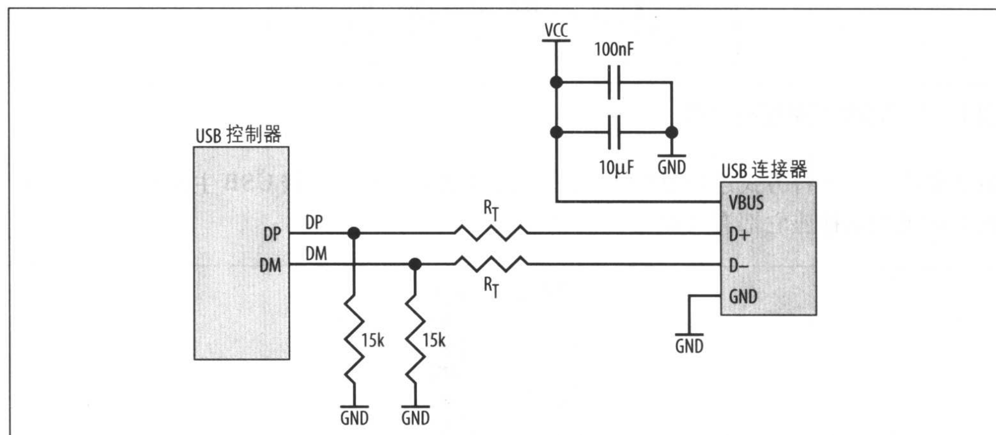


图 11-11：下行 USB 端口

在以上两个图中，**DP** 和 **DM** 都有串联电阻（R_T），串联电阻将 USB 连接终止。总的电阻值应该为 45Ω，但是 USB 控制器的引脚本身也有阻抗，因此都要考虑在内。如果引脚的阻抗是 21Ω（假设），那么串联电阻就应该为 24Ω。这里需要注意的是，并不是所有的生产商都会在他们的技术手册里把引脚的阻抗完全写清楚，在这种情况下，可以向生产商索取这些数据或者赌一把，串联电阻值的范围应该在 20~30Ω 之间。

许多微控制器，比如Microchip公司的PIC16C745和PIC16C765都包含USB模块，作为其I/O组的一部分。用这些处理器实现USB很容易，只需将物理接口连到处理器的**DP**和**DM**引脚即可。然而，如果选择的芯片是作为基本嵌入式处理器来使用的，没有包含USB，你就必须用外部设备来提供USB功能。

USB接口的实现

在嵌入式系统中实现USB一个可能的方法是用USB-to-SPI桥接器，比如Atmel AT76C711。这个芯片是一个带USB子系统的AVR处理器，被设计用来担任主机处理器的从USB控制器角色，它有2K数据RAM、2K用于包处理的双口RAM、16K程序RAM（组织形式为8K×16字）、1个内置的DMA控制器、1个上行USB端口（有1个控制、5个数据终端）、1个独立的兼容IrDA的UART和SPI。这个处理器运算速度可达24MHz，工作电源为3.3V。复位的时候，AT76C711自动从一个外部AT45DBxxx数据闪存（参见第7章）将它的软件装入程序RAM。由于AT76C711的程序空间相当小，因此AT45DBxxx系列的数据闪存芯片中比较小的1个就足够了。换句话说，在复位时主机处理器会将AT76C711中的代码直接装入它的程序RAM中。

AT76C711可以单独作为一个处理器，完成从USB到RS-232C、RS422/RS-485、IrDA或其他协议的桥接功能。换句话说，它可以内置在一个较大的嵌入式系统中作为一个从处理器。通过SPI总线或者是使用AT76C711的UART的一种标准串行接口，主机处理器可以与AT76C711通信。AT76C711还有通用I/O线和一个UART模块，它们支持用于IrDA的RZ编码。

如果使用的处理器有个总线接口，那么你可以使用芯片，比如Agere Systems公司（*http://www.agere.com*）的USS-820D来添加USB。这个芯片支持的传输速率可达12Mbps，并且不像许多USB芯片那样用于集线器中，它是专门设计用在USB设备中的。USS-820D可以支持的端点数达8个，每个都有超过1120字节的接收和发送缓冲器。

图11-12是一个使用USS-820D的上行USB接口示意图。

USS-820D有几个电源输入端（V_{DDA}、V_{DDT}、V_{DD0}和V_{DD1}），它们都接到一个3.3V的电源（图11-12中的**VDD**）。每个电源引脚都用一个100nF的电容接地来退耦，从USB连接器得到的5V电源（V_{BUS}）不能够直接用来驱动USS-820D，不过一个MAX604电压调节电路（参见第5章）可以将V_{BUS}转换为要求的3.3V电源。USS-820D还有接地引脚（V_{SST}、V_{SSX}、V_{SS0}、V_{SS1}和V_{SS2}），它们都和地线相连。即使USS-820D使用3.3V的电源，它的数字（非USB）输入也是与5V电平逻辑兼容的，因此这个芯片可以直接与工作在5V电源下的处理器相连。

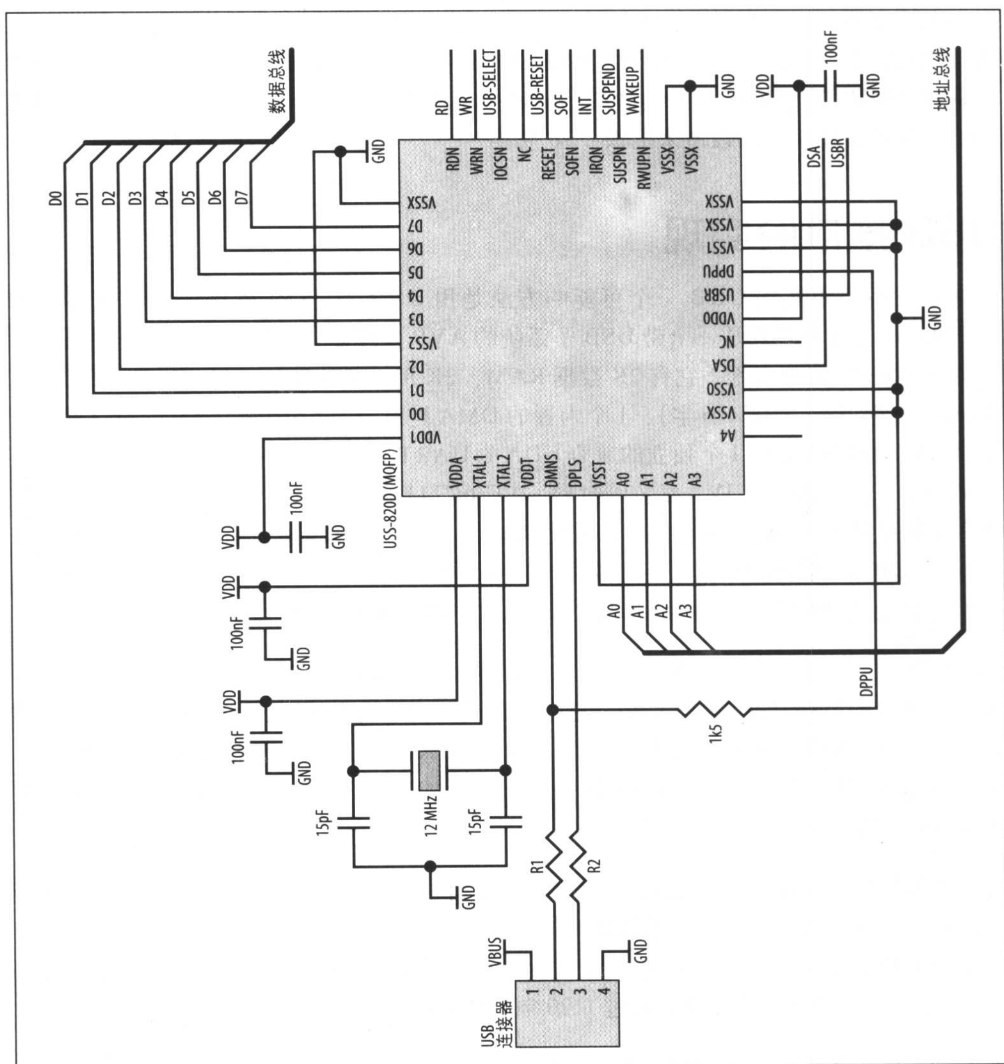


图 11-12：USS-820D USB 接口

XTAL1 和 **XTAL2** 是连接一个 12MHz 晶体上的两个引脚，这个晶体向 USB 控制器提供时钟功能。

到微处理器的连接很直接，这个设计可以包括在第 15 章中基于 AT90S8515 的计算机设计中。USS-820D 的数据引脚 **D0~D7** 直接和处理器的数据总线相连，地址引脚的低 5 位 **D0~D4** 和处理器相应的信号线相连，这些地址位是用来选择 USS-820D 中的内部寄存器

的。处理器的读（**RD**）和写（**WR**）信号直接与USS-820D的读（**RDN**）和写（**WRN**）引脚直接相连（Agere公司在低电平有效的引脚名字后面加了一个N）。当**IOCSN**被置为低电平时，就选中USS-820D，因此这个引脚是由地址译码器的一个输出（在图11-12中标为**USB-SELECT**）来驱动的。

当芯片的**RESET**引脚为高电平(不像大多数其他设备一样是低电平)时，就将USS-820D复位。因此，对于普通工作模式，这个引脚应该保持低电平。要想允许USS-820D可以在软件控制下复位，这个引脚可以由处理器的一根数字输出线来驱动。

USS-820D有许多输出，可以用来告诉主机处理器当前的USB状态。**DSA**（数据设置可用）、**USBR**（USB复位探测）、**SUSPN**（挂起）和**SOFN**（帧开始）都可以被主机微控制器作为数字输入来读取，或者对于有几个中断输入的处理器来说，这些信号可以用来产生中断。所有这些事件都会触发USS-820D的中断引脚（**IRQN**），因此如果主机处理器的中断能力有限，这个引脚可以作为连接处理器的唯一中断引脚。有一点要注意：在软件控制下可将这个引脚设置为高电平有效，也可以设置为低电平有效。如果将这点搞错了，嵌入式系统就会一直处于中断状态。这个引脚默认的状态是低电平有效，这种设置对于大多数处理器都适用。对于那些中断输入高电平有效的处理器，比如Intel处理器，必须在处理器的中断处理能力有效之前用固件将USS-820D设为正确的中断极性。

引脚**RWUPN**是一个远程唤醒输入信号，也就是说嵌入式系统曾经睡眠(处于挂起模式)而现在已经“醒”过来。这个引脚向USS-820D通报状态的改变，这样USS-820D就可以通知其他的USB系统。**RWUPN**只是通过处理器的一根数字输出线来驱动的。

USB差分数据信号是指引脚**DPLS**（数据正，**D+**）和**DMNS**（数据负，**D-**），这两个引脚通过串联终结电阻和USB连接器相连，Agere Systems公司建议这个电阻的标准值为24Ω。对于一个上行连接，DPLS（D+）需要一个1.5kΩ的电压上拉电阻，通常情况下这个电阻连到5V电压，不过USS-820D有个专门用于这个目的的特殊引脚（**DPPU**）。因此，在软件控制下，USS-820D可以模拟一个USB设备的断开，即使还连接着，上行集线器还是会看到包含USS-820D的系统已经在物理上断开了。这点在开发和测试期间非常有用，它还可以让USB设备决定是否让主机知道自己连接了。**DPPU**可以通过一个10nF的电容接地来退耦。

为嵌入式硬件添加USB功能时，使用USS-820D之类的芯片，可以使工作变得简单和容易。通过使用USB，可以为台式机系统开发基于嵌入式处理器的外设。也就是说，可以用USB连接嵌入式系统和已经存在的外设，以进一步增加系统的功能。

第12章

网络

永远都不要为将来所困扰。
你终将面对未来，
到时，就如现在面对现实那样去面对它。
——马可·奥里利乌斯
《沉思录》

没有一个城镇或自由人应被强迫着去建造桥梁……
除了那些负有历史使命需要这样做的人。
——《大宪章》

在这一章里，我们将看到通过增加一个局域网（Local Area Network，LAN）接口来把你的嵌入式计算机和现实世界连接起来。现在使用的网络种类繁多，它们有些很常见，有些不很常见。我们接着分别看一下控制器局域网络（Controller Area Network，CAN）和以太网（Ethernet）。

CAN适用于电气干扰和条件状况较差的地方，是一种在电气恶劣的环境中可以选择的网络。以太网是企业内部互联网络（Intranet），它可以连接世界上所有的台式计算机以及其他一些设备，如路由器、网关、打印机和另外一些外部设备。

CAN

自20世纪70年代末期到20世纪80年代整个时期，汽车电子、引擎管理系统、ABS刹车装置、智能型悬架装置、电子通信设备和自动照明、空调、安全系统（装置）及中央门锁等的复杂性已经有相当程度的增加。这些单个系统都不是孤立存在的，它们都是一个综合系统的一部分。这就要求有一定数量的信息交换，因此也就必须提供一些系统间

的相互连接途径。传统的方法是点对点连线，来分别提供各个子系统间的相互连接。这种方法是从早期的汽车电子系统很自然地发展而来的，但是由于汽车复杂性的增加，这样的一个系统也就在很大程度上不能满足要求了。每辆汽车可能有几公里长的电路线和几十个连接器，诸如此类的布线系统大大地增加了生产一辆汽车的成本，增加了不必要的重量，降低了安全性，并且使得维修变成一项可怕的工作。

对于上面这种情况，一个很显而易见的解决方法是以简单来替代复杂，用一个低成本的数字网络来实现系统间的通信。汽车的电子环境干扰很严重。在采用电动引擎、电子打火、电子射频散热等等的情况下，向汽车的电子系统输入12V的电压可以产生上400V的瞬间高压，这就要求通信网络必须能够适应这样的干扰环境并且稳定地工作。因此这样的网络必须具有很强的抗干扰、错误检测和处理能力，可以重发失败的数据包。正是在这种需求下，出现了控制器局域网络（CAN），它在双线串行网络里的实时通信速率高达1 Mbps。CAN只规范了ISO-OSI模型的物理层和数据链路层，其他更高层则在具体实现时根据不同的情况来完成。

在20世纪80年代末，Bosch在欧洲开发出了CAN，最初是用在汽车上的。由于其健壮性，CAN的应用范围已远超出最初的汽车应用，现在，在工业自动化、火车、轮船的导航和控制系统、医疗系统、影印机、农业机械、家用电器、办公自动化和电梯中都可以看到CAN。CAN目前已经成为ISO11898和ISO11519-2下的一个国际标准。

CAN在网络中支持多主机，在整个分布式系统中每个主机负责各自局部的监听和控制(如图12-1所示)。

每个CAN信息包都包含地址信息和优先级作为信息包头的一部分，所有的节点可以连到网络上，也可以从网络上断开，但不会影响网络上其他节点的数据传输。

CAN采用wired-AND逻辑，最大总线长度可达1000米，当以最大的速率在双绞线上传输数据时，总线长度可达40米。总线的每个末端都要求有终端电阻，以避免传输信号反射（如图12-2所示）。

许多用于条件恶劣或电子干扰存在的工业应用的处理器都包含一个CAN模块。如同少许PIC单片机一样，许多Philips微控制器都包含有CAN。我们将在第19章中介绍的DSP56805处理器也有一个CAN接口。虽然有些处理器不包含CAN，但却有CAN接口模块可用。Microchip MCP2510提供了一个CAN模块，这个模块可以通过SPI接口连接到一个主机处理器上。因此，将CAN添加到任何一个嵌入式系统中都是一件非常简单的事情。

一般情况下，一个支持CAN的微处理器会包含一个CAN接口模块，这个模块提供了绝大多数的功能，唯一需要增加的支持是一个CAN接口驱动器（就像RS-485和RS-232C

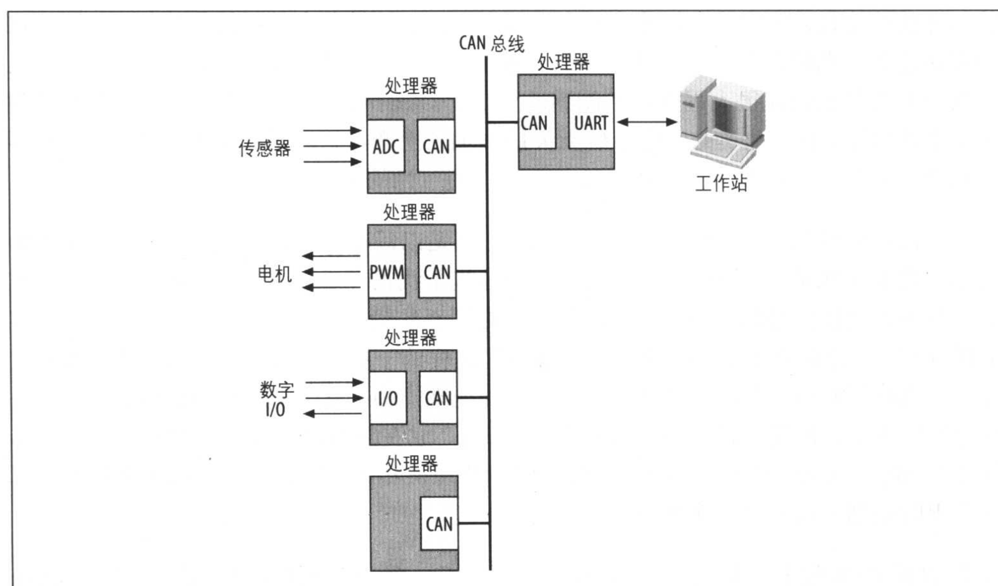


图 12-1：CAN 分布式系统

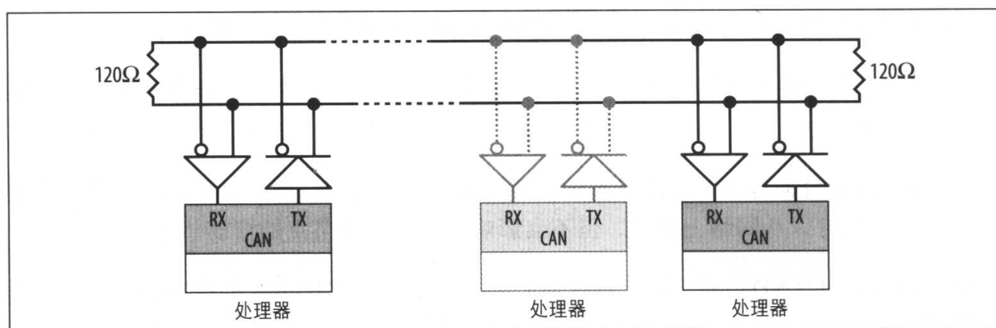


图 12-2：CAN 总线

中的那样）。Philips 半导体公司生产了一种 CAN 驱动器：PCA82C250T，这个产品使得连接到 CAN 总线非常容易。

你的嵌入式计算机也必须要有通过物理连接到 CAN 总线的某种方式。最简单的方法是将总线接上计算机系统的一个连接器，将总线引入，然后通过另外一个连接器将总线引出（如图 12-3 所示）。

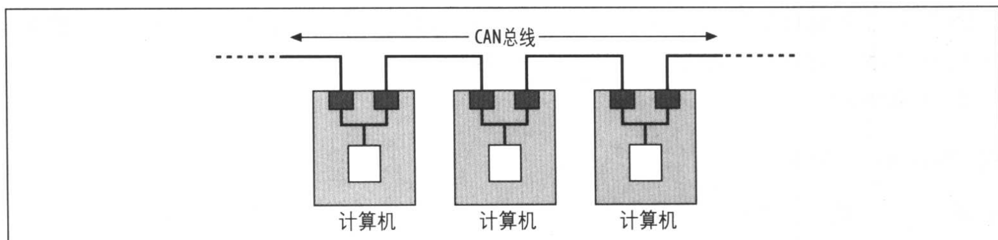


图 12-3：使用 PCB 上的两个连接器接入 CAN 总线

为了弄明白如何使用CAN，先来看一下DSP56805处理器。这个处理器有一个CAN网络模块作为其携带式外围设备组的一部分。图12-4显示了如何连接一个微处理器的CAN模块到一个CAN总线。

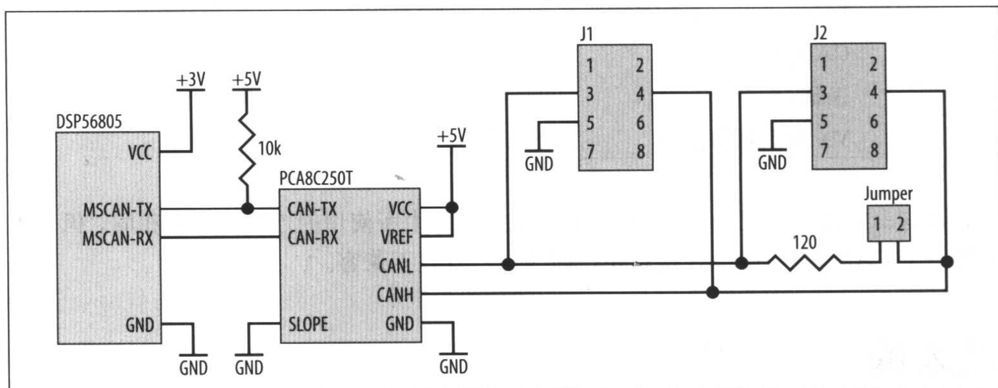


图 12-4：一个 DSP56805 处理器的 CAN 接口

DSP56805有两个CAN接口信号：**MSCAN-TX**和**MSCAN-RX**，分别是CAN的发送器和接收器。**MSCAN-TX**和**MSCAN-RX**连接到PCA82C250T，而PCA82C250T提供了连接到CAN总线的接口。注意，DSP56805要求一个3.3V的输入电压，然而PCA82C250T却要求5V的输入电压。用一个电压上拉电阻可以将处理器**MSCAN-TX**的输出变为PCA82C250T要求的逻辑高电平。CAN只要求两条信号线，而实际中的连接器有8个引脚。由于CAN总线要求在每一端有一个终端电阻，如果我们要将计算机放到总线末端，我们就提供一个120Ω的电阻。通过一个跳线可以根据需要将计算机置为在线或禁止。所以，如果我们的计算机被置于CAN总线的末端，一旦跳线关闭，那么总线就终止了。如果我们的计算机不是一个末端机器，跳线置于断开，这个电阻就不起作用。记住，当一个计算机不是处于末端时，将一个终端电阻置于有效（跳线闭合）是确保一个不可靠CAN总线正常工作的好方法。电阻应该只在总线末端才处于有效状态。

CAN的许多实现只使用了标准IDC（智能磁盘控制器）类型的接头作为连接器。然而，实际的CAN标准规定连接器应该是一个9引脚的Sub-D连接器。这种连接器的引脚如表12-1中所列。

表12-1：CAN引脚

引脚	信号/作用
1	保留
2	CAN_L
3	接地
4	保留
5	保留
6	接地
7	CAN_H
8	保留
9	V+（可选择电源）

尽管这与一些用于实现RS-232（如PC中的串口）中所使用的连接器类型相同，但是不要把一个CAN和RS-232连接起来，因为这二者并不兼容。

以太网

任何一个人，即使与计算机关系甚远，只要他（她）接触了计算机，就一定听说过以太网。20世纪70年代早期在施乐公司PARC（注1）研究中心开发出来的这个局域网标准已经在每一个可能的应用中找到了它的位置，而且随着时间的不断推移，已经包含从无线网络（802.11）到千兆以太网的许多标准。

在这一部分，将向你展示如何将一个简单的以太网接口添加到你的嵌入式计算机中。我们只是要开发一个10Mbps的接口，因为更高速度的接口需要特别注意PCB设计和EMC（电磁兼容性）问题。所以，为了容易和可靠，我们将选择简单和低速的接口。

以太网标准和协议的详细内容可以从Charles E. Spurgeon所写的《Ethernet: The Definitive Guide》一书中得到，该书已由O'Reilly公司出版。这本绝佳的书籍对以太网

注1： PARC即Palo Alto Research Center（*http://www.parc.com*），有关PARC（和计算机工业大体上）的有趣历史，请参阅Robert X. Cringley的《Accidental Empires》一书。

进行了准确详细的论述，是每一个从事基于以太网硬件开发的人员不可缺少的一本书，也是一本入门的读物。

通过将以太网添加到你的嵌入式计算机系统中，你就可以连接到一个网络，并访问这个网络所有可能带来的东西。你可以向一个主机高速发送数据并访问打印机、文件服务器、数据库，甚至Internet。你还可以检测和控制远方的嵌入式系统，或者，甚至可以令它在其可能出问题的时候向你发送电子邮件。拿一块AT90S8515 AVR微处理器，再增加一个以太网接口和一些大容量的闪存，就拥有了一个简单的Web服务器。然后，再添加一个模数转换器（ADC）和一些传感器，你的Web服务器就变成了一个气象站，可以向Internet上的任何一个人显示当前或过去的气象状况。使用一个高速的处理器、几个以太网端口和合适的软件，你就有了自己简单的网关或防火墙。你甚至可以构建一个Ethernet-to-Ethernet（或串口、并口、USB）的网桥。只要你能想象得到的都是可能实现的。

曾经有一段时间，开发一个以太网接口是一个主要的练习内容，它们都是一些复杂的电路，使用大量的芯片和几百个支持部件。一个以太网接口，别的什么都不需要，就可以填满一个中等规模的PCB。在大规模集成电路发展的今天，在你的设计中增加一个以太网接口就非常简单了，接下来我们将会看到。

添加一个以太网接口

Crystal Semiconductor公司现在成为Cirrus Logic公司（*http://www.cirrus.com*）的一部分，生产了一种单芯片的以太网控制器CS8900A。这种芯片允许你向嵌入式系统中增加一个简单的（或低速的）10Mbps以太网接口。关于这块芯片的完整文档可以从Cirrus Logic公司的网站上得到。由于CS8900A是一个常用的以太网控制器，因此从Internet上还可以得到足够的源代码。用你喜欢的搜索引擎来搜一下看看！当设计一个基于CS8900A的系统时，你可以将你的设计方案通过电子邮件发给Cirrus Logic公司的工程师，他们会帮你检查，为你提供建议并指出错误的地方。提供这个服务的电子邮件地址是：ethernet@crystal.cirrus.com。

CS8900A支持*10BASE-2*、*10BASE-T*和*AUI*（Attachment Unit Interface，连接单元接口）以太网端口。10BASE-2和10BASE-T是目前最常见的以太网接口类型，它们的数据传输率分别为10Mbps和100Mbps。你的台式计算机的以太网接口很可能就是一个带有8个引脚的RJ-45连接器的10/100BASE-T端口（RJ-45连接器看起来像——不过不是——标准的电话插孔）。电缆连接使用的是*UTP*（Unshielded Twisted Pair，非屏蔽双绞线）5类电缆，更常见的叫法是*CAT5*。就像RS-422、RS-485、USB和CAN一样，10/100BASE-T传输采用平衡差分信号。使用四根信号线，其中两根用来发送，其

他两根用来接收。两对信号线中的一根承载 0~2.5V 的信号电压，而另一根负载的电压是 0～ −2.5V，因此就可以产生一个 5V 的信号差。

表 12-2 给出了 RJ-45 连接器的引脚说明。CAT5 电缆中的信号线通过颜色来编码，很好辨认。

表 12-2：RJ-45 连接器信号

引脚	信号名称	用途	颜色
1	TD+	发送数据	白 / 橙色
2	TD-	发送数据	橙色
3	RD+	接收数据	白 / 绿色
4	NC	不连接	蓝色
5	NC	不连接	白 / 蓝色
6	RD-	接收数据	绿色
7	NC	不连接	白 / 棕色
8	NC	不连接	棕色

图 12-5 所示是一个 CS8900A 的实现框图。

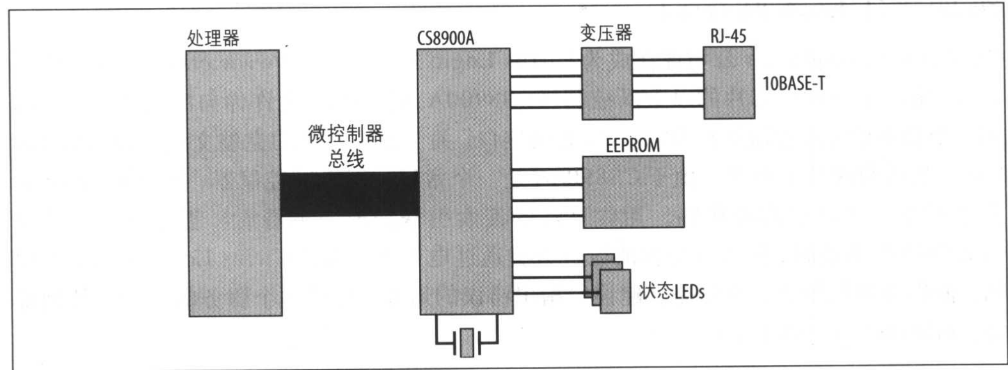


图 12-5：一个 CS8900A 的实现框图

由于CS8900A有100个引脚和几种不同的操作模式，所以我不打算在这里把全部引脚的说明都列出来，而是随着介绍CS8900A的每个设计阶段将其功能和用法逐步加以解释。这个说明过程针对的是一个小型嵌入式应用。CS8900A一些应用在台式计算机中更复杂的方面在这里不再提及。

CS8900A通过一个隔离的变压器连接到计算机的10BASE-T端口上。如果CS8900A使用的是5V的电源，则这个变压器必须有两个绕线比为1:1和1:1.41的线圈分别供接收器和发送器使用。如果CS8900A的供电电源为3.3V，那么变压器供发送器使用的线圈的绕线比必须为1:2.5。许多生产商如Valor、PCA、YCL以及Bel都生产这类绕线比的隔离变压器（像芯片一样包装）。发送器要求电值为24.9Ω（1%）的串联终端电阻，而且发送器的两个差动端必须使用一个68pF的电容来相互退耦。在接收器的两个差动端需要通过一个100 Ω（± 1%）的电阻进行连接。CS8900A还可以直接驱动LED，这些LED可以指示以太网的链路状态以及总线和网络的活动情况。CS8900A另外还有一个引脚（RES），这个引脚需要接一个4.99kΩ（±1%）的降压电阻。图12-6所示为CS8900A连接到一个10BASE-T端口的示意图。

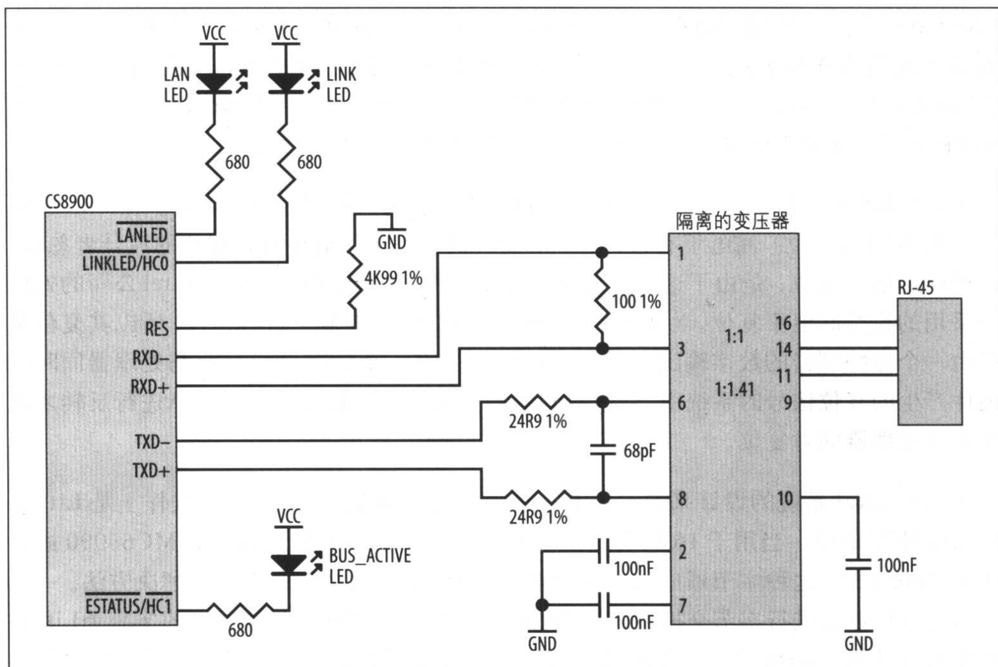


图12-6：BASE-T接口

一个外部20MHz的晶体为CS8900A提供时钟。这个晶体连在两个引脚XTAL1和XTAL2之间，这两个引脚分别通过一个33pF的电容接地（如图12-7所示）。

CS8900A芯片支持16位的ISA总线结构，其扩展总线在一些老式PC机中可以看到。然而，ISA能够很容易地和一部分非ISA处理器兼容，因此CS8900A可以毫不费劲地在许多计算机系统中使用。同时CS8900A还支持在8位模式下的操作，所以能够与带8位数

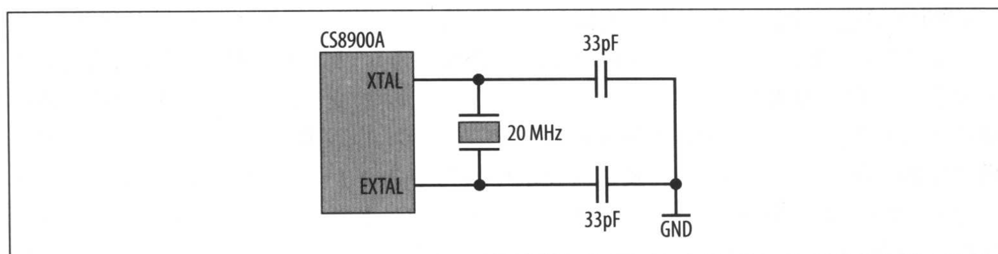


图12-7：晶体与CS8900A的连接

据总线的微控制器相连，如AT90S8515 AVR微处理器。AT90S8515的输入引脚**SBHE**用来在复位后将该芯片置于16位操作模式，而且**SBHE**引脚上的任何有效输入都会将CS8900A置于16位操作模式。确保**SBHE**引脚的输入有效最简单的方法是将其接到处理器地址线的**A0**线上，这样一旦处理器开始使用其总线，**SBHE**的有效输入将会置CS8900A于16位模式。如果想让CS8900A处于8位模式，需要将**SBHE**接地。当使用8位模式时，中断都被禁止，这时CS8900A的状态必须由软件来设定。

在了解CS8900A的处理器接口之前，我们需要注意其一些很重要的特性。在CS8900A上，**RESET**（复位）高电平有效，这点可能会被习惯于低电平有效复位的设计者忽略。之所以高电平复位，是由于这个芯片是设计用来供PC机使用的，而且Intel公司的处理器采用的也是高电平复位。为了使CS8900A可以在软件控制下进行复位，所以其复位可以由一个微控制器的数字输出来驱动。换句话说，在一个CS8900A接收与处理器同时的硬件产生的复位信号的系统中，处理器的低电平复位信号必须为CS8900A进行反转才能使其余处理器同时复位。

对于基于Intel系统的设计来说，还有一点要注意，就是CS8900A在操作上是Little-Endian字节顺序。当用于16位模式Big-Endian字节顺序的处理器（如MC68000或者DSP56805）时，这种字节顺序的差异就很重要了。对于这个问题有两种解决方法。一种是只通过软件来进行字节交换，程序在将数据写到CS8900A之前将16位的字变为Little-Endian字节顺序的格式，而在从CS8900A读数据的时候，处理器必须在处理数据之前将其进行字节交换。

然而，有一个古老的说法：能够在硬件中改正的东西永远不要拿到软件里来修改。因此也就有了第二个解决方法：在CS8900A和处理器之间对数据进行字节交换，将处理器的**D0:D7**和CS8900A的**D8:D15**相连，而将处理器的**D8:D15**和CS8900A的**D0:D7**相连。这样就将字节顺序问题在电路板上解决了，软件永远没有必要知道其中的差异（如图12-8所示）。

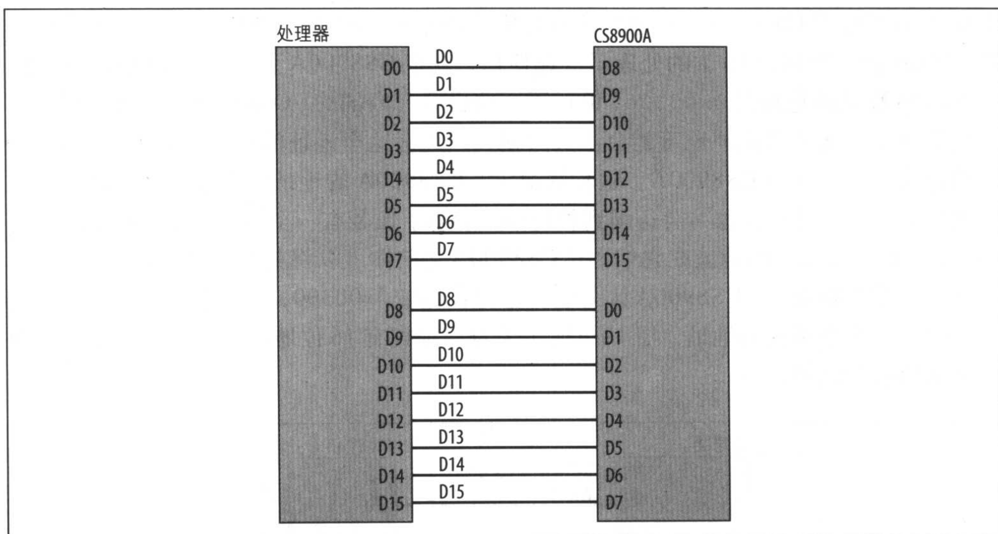


图 12-8：通过硬件进行字节顺序交换

CS8900A有20个地址输入引脚，这看起来就像是为外围设备所准备的许多地址输入，实际也确实如此。然而，这里有一个原因。CS8900A主要是一个ISA总线设备，而ISA总线支持单独的内存空间和I/O存储空间。因此，CS8900A就有两个单独的处理器接口。在一个端口里，这个芯片看起来就像是一个处理器内存空间的一部分，并且可以像访问一个存储设备那样被访问。选择输入引脚**CHIPSEL**的作用就是当CS8900A用作一个内存映射设备的时候使其有效。当CS8900A作为一个I/O空间中的设备时，不需要外部的选择输入来使其有效。而且，通常认为被映射到ISA总线I/O空间中的设备都有自己的地址译码部分，这也就是为什么CS8900A会有20根地址线的原因。在CS8900A里，有一个专门的地址译码器。当它复位时，就还原为默认的I/O地址 `0x00300`。在软件控制下，这个地址可以通过写CS8900A适当的寄存器进行重新映射。当CS8900A作为一个I/O映射设备时，**CHIPSEL**被忽略，而且在**IOR**（I/O读）和**IOW**（I/O写）的协同工作下，它还会对其地址输入端输入的有效地址产生响应。你可以在一个内存映射I/O系统中在I/O模式下使用CS8900A。该系统的地址译码器负责为CS8900A分配地址，但是并不选定它。系统的地址译码器必须要做的是确保当访问对应于CS8900A的地址时，不是别的设备被访问。

CS8900A的默认设置是I/O模式操作。要想在内存映射模式下使用它，并因此可以识别**CHIPSEL**、存储器读（**MEMR**）和存储器写（**MEMW**）输入，CS8900A必须首先作为一个I/O映射设备被访问，并且通过软件进行重新配置。因此，要想使用内存映射功能，你还必须支持I/O映射寻址电路才能实现。也就是说，保持在I/O映射模式，按前

面描述的那样将 CS8900A 的地址映射到你的内存空间中要简单得多。如果你正在将 CS8900A 和一个只有 16 位的处理器一起使用，就把 CS8900A 多余的地址线直接接地。CS8900A 默认的地址是 0x00300，但对于一些已经有内部的 I/O 系统映射到这个区域的处理器来说，这个默认地址可能会不太方便，因为对这个地址的访问将会被内部的 I/O 拦截而永远不会到达 CS8900A。这种情况下，CS8900A 的地址就不能够通过软件来重新映射，因为你将永远也不会访问到相应的寄存器。但是有一个解决方法，是在硬件层的。如果处理器发出的地址在它们到达 CS8900A 之前你可以将某些位进行取反，就可以自动进行重新映射了。CS8900A 认为它自己仍在地址 0x00300，而对于处理器来说则是在访问一个完全不同的地址。图 12-9 所示为对一个拥有 16 位地址总线的处理器进行地址重新映射的例子。

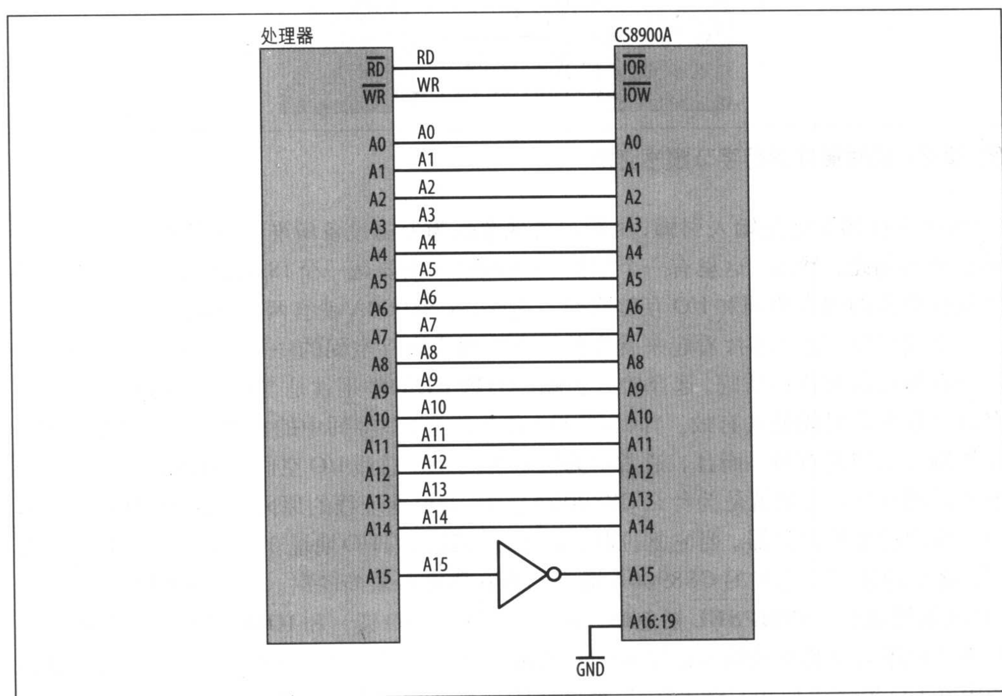


图 12-9：在硬件层进行地址重新映射

在这个例子里，地址位 **A15** 取反。所以，当处理器访问地址 0x8300（%1000 0011 0000 0000）时，这个地址就被转换为 0x0300（%0000 0011 0000 0000），而后者可以被 CS8900A 所识别。

CS8900A 还支持一个串行 EEPROM。这个存储器可以用来保存 CS8900A 的配置信息和系统唯一的物理地址。注意，这个 EEPROM 是可选的，因为主机处理器可以将这些信

息保存到系统的其他地方。图 12-10 所示是 CS8900A 和一个配置 EEPROM 的连接示意图，所使用的接口是标准 SPI，CS8900A 的引脚与 EEPROM 相对应的引脚直接相连，唯一需要增加的部件是一个接到 EEPROM 的电源引脚上的退耦电容。EEPROM 的接口被禁止设为 8 位模式，所以主机处理器必须提供所有的配置信息。

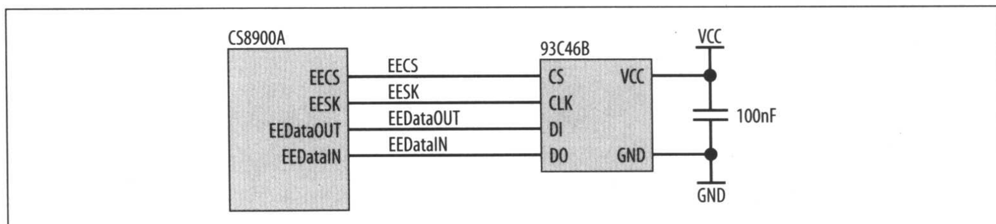


图 12-10：CS8900A 和一个配置 EEPROM 相连接

最后，对于可能出现的输入，比如 DMA 信号（**DMACK0**、**DMACK1** 和 **DMACK2**）、**TEST**、**SLEEP**、**MEMW**、**MEMR**、**AEN** 和 **REFRESH** 都应该置于无效，因为在一个典型的嵌入式系统中不使用这些信号。

第13章

模拟量

常无欲，以观其妙；
常有欲，以观其徼。
此两者，同出而异名。
——老子
《道德经》

在这一章，我们将会看一看如何对外部的模拟电压进行采样，并将它们转换成嵌入式系统可以处理的数字值。这些电压可能是由传感器产生的，代表光的强度、温度或振动，也可能是麦克风或音频系统的输出电压，需要被转换为数字信号。稍后，我们将会了解如何将数字信号转换为模拟的输出电压。在本章的最后，将讨论一下关于控制电机的硬件。

首先，让我们快速地了解一下放大器和采样原理。注意，这是一个非常复杂的领域，涉及到的基本原理已经大大超出了本书的范围，所以这里只作大致的介绍，以使你能够有足够的背景知识来将嵌入式系统和简单的模拟电路连接起来。由于我们有意地简化讨论，因此这个介绍并不详尽彻底，而且被刻意地简单化了。

放大器

放大器用来连接两个模拟电路。放大器是一个可以对一个输入电压进行提升（或降低）以产生一个新的输出电压的电路。比如，你现在有一个最大输出只有5mVpp的传感器，而这个传感器被接到一个要求输入信号为5Vpp的采样系统。这时你就得在传感器和采样系统之间使用一个放大器，来将传感器的输出电压进行相应的提升（如图13-1所示）。

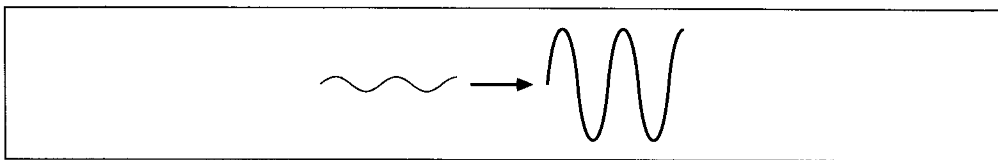


图 13-1：放大一个波形

放大器的输出波形应该和输入信号的波形一样，发生变化的只是振幅。信号提升或降低的数量被称为放大器的增益(gain)。增益的计算很容易，用输出电压除以输入电压即可：

$$Gain = V_{OUT} / V_{IN}$$

所以，一个将输入信号加倍的放大器，其增益为2。

电路对一个变化的信号的响应能力通常局限于一定的频率范围，这称作电路的频率响应。比如，你的家用立体声设备里的放大器的频率响应可能是20Hz～20kHz，这意味着它只对处于20Hz（低音）~20kHz（高音）之间的音频信号进行放大。如果你试着将一个100MHz的信号输入音频电路，它就不能够对这个信号进行放大，因为该信号超出了它的频率范围。

理想情况下，一个电路（如音频放大器）的频率响应在其频率范围内应该是平坦的。也就是说，不管频率是多少（在频率范围之内），它对输入信号的响应应该相同。所以，在音频放大器的例子里，对于在频率范围内任何信号的增益应该是相同的，通俗地讲就是音量将不会随着频率（忽略任何由于原始音乐的原因而带来的不同）的改变而改变。在频率范围的两端，放大器不能理想地工作，其增益会降低，这种情况被称为转降（roll off）。通常认为比较小的转降是可以接受的（并且是不可避免的）。频率响应是定义在一个频率范围上的，在这个范围内，增益处于一定的理想限度之内。

放大器对输入信号放大后在输出端输出时的主要局限是失真。你经常会在音频放大器的说明书中看到一个参数：总谐波失真（Total Harmonic Distortion，THD）。这个值越小，放大器性能就越好。

过去，放大器是用离散的晶体管（注1）或者真空管（也称作电子管）构成的。现在，放大器已经被集成在集成电路里面，这些放大器被称作运算放大器，或简称为OP放大器。它们便宜、可靠，且非常容易使用，从而使设计者的工作变得简单多了。在本章中，当需要放大一个电路的时候，我们都会使用一个OP放大器来实现。OP放大器的电路原理图符号如图13-2所示。

注1：在某些特别的应用里，放大器仍可能采用离散的晶体管（或者甚至电子管）。

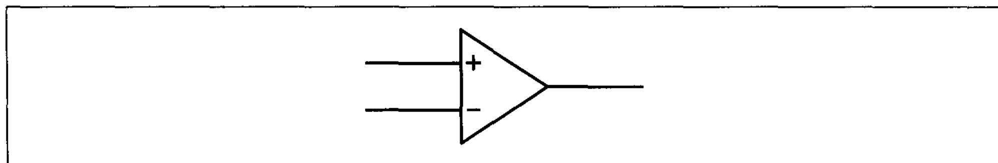


图 13-2：OP 放大器的电路原理图符号

标“+”的输入端称为正相输入端，标“–”的输入端称为反相输入端。如果在正相输入端输入的电压大于反相输入端的电压，OP放大器的输出为正；反之，输出为负。通常情况下，由于内部电路的限制，OP放大器的输出电压将不会像其负电源的值那么低，也不会像正电源的值那么高。当一个OP放大器的输出电压处于正电源与负电源之间时，我们称这个放大器具有 *Rail-to-Rail* 的运算功能。

为了正确地工作，OP放大器需要反馈。反馈需要将OP放大器的输出接回输入端。负反馈利用输出来减小放大器的增益，籍此提高放大器的其他性能，比如频率响应的平坦程度和抗干扰能力。负反馈的实现很简单，只需在反相输入端和输出之间增加一个电阻就可以了，我们稍后就会看到。没有反馈的电路称为开环。通过一个反馈电阻，OP放大器可以利用输出的改变来消除与输入之间的差异。因此，输出波形随着输入波形之间差异的改变而改变，而输出波形幅值的大小和反馈电阻成比例。这个电阻越大，输出的反馈量就被削弱地越多，为此，OP放大器就加大输出来进行补偿。这样，输出就成为输入放大后的结果。

一个 OP 放大器可能会被用作反相放大器（如图 13-3 所示），也可能会被用作同相放大器（如图 13-4 所示）。反相放大器在对信号放大的同时也将其反相。

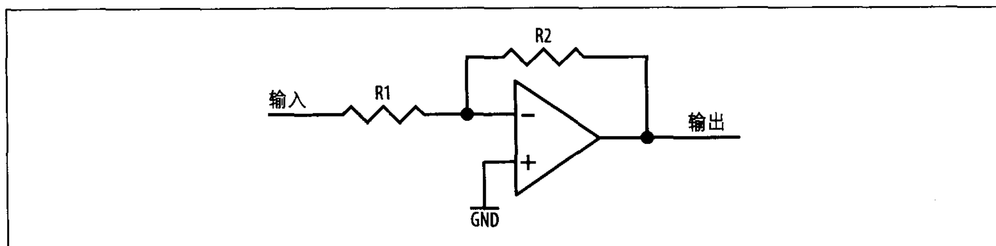


图 13-3：反相放大器

反相放大器的增益由下式给出：

```
Gain = -R2 / R1
```

注意上面的负号，因为反相放大器将信号反相了。

你可能更喜欢使用同相放大器，因为它不会将信号反相。这种放大器通常用在音频和传感器应用中。

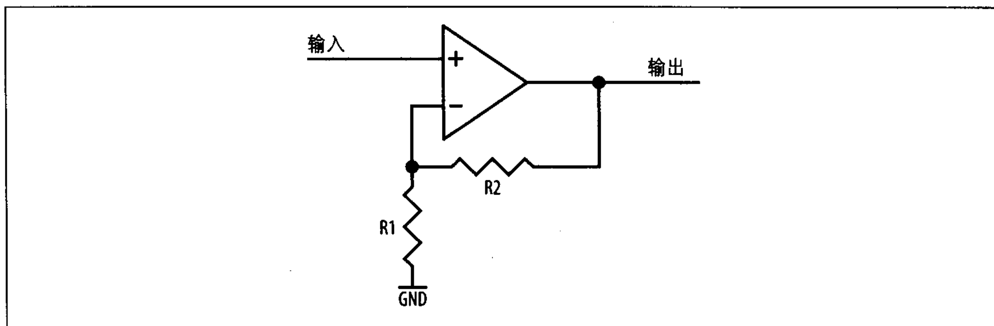


图 13-4：同相放大器

同相放大器的增益由下式给出：

```
Gain =1 + R2 / R1
```

放大器的增益可以使用一个数字电位计来替代R2在软件环境下进行设置（参见第7章）。

差动放大器（如图 13-5 所示）用来将两个输入信号之间的差异增大，并且放大容易受到噪声干扰的小信号。通过将感兴趣的信号和基准电压之间的差异放大，任何噪声的影响都变小了（由于噪声不但会影响信号，而且同样会影响基准电压）。当差动放大器的两个输入端变化相同时，被称作共模（common-mode）变化。理想情况下，差动放大器是不会出现共模变化的，因为其目的就是用来放大信号之间的差异。对共模变化的抑制性称为放大器的共模抑制比（Common-Mode Rejection Ratio，CMRR）。CMRR 越高，放大器就越好。为了得到高CMRR，尽可能使匹配电阻的值（和偏差）接近就显得尤为重要。

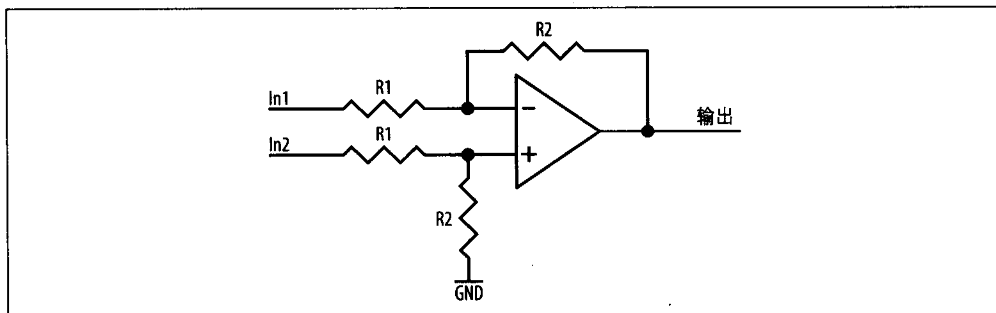


图 13-5：差动放大器

差动放大器的输出电压由下式给出：

$$V_{OUT} = (In2 - In1) * (R2 / R1)$$

模/数转换

将一个模拟输入电压转换为一个数字的设备称为模数转换器（Analog to Digital Converter，ADC）。也许你以前还听说过编码－译码器（COder-DECoder），编码－解码器将一个ADC和一个数模转换器（Digital to Analog Converter，DAC）相结合，在同一块芯片上既提供模拟输入也提供数字输入。在本章后面部分将更详细地介绍DAC。

可以在移动电话和数字电话中看到ADC，它们将声音转换为数字数据来发送。在计算机中也有ADC，它们在声音识别时对麦克风的输入信号进行数字化。专业录音室使用ADC来将音频转换为数字数据来供制作原版CD使用，与此相似，视频是在制作原版DVD之前先用ADC进行采样。扫描仪、网络摄像机（web cam）和数字摄像机里也都有ADC。在其他应用领域里，ADC用来对传感器的输入进行采样。从自动气象站到PC机里处理器温度的系统监控，都可以看到这些应用。

ADC有几种不同的类型。集成ADC使用一个内部由电压控制的振荡器来产生时钟信号，这个信号的频率与采集到的电压成比例。时钟信号用来驱动一个计数器，采样电压的数字值就由这个计数器提供。采集到的电压越高，时钟频率就越高，因此计数器达到的数也就越大。计数器在每次转换之前都要进行复位，因此集成ADC的转换速度极低，不为人们所熟悉。

逐次逼近ADC使用一个DAC来提供一个参考电压，这个电压与输入电压进行比较。当增加驱动DAC的数字码时，参考电压也会增加，直到找到一个与输入电压匹配的值。一旦匹配成功，驱动DAC的数字码就是ADC的数字输出。

高速ADC（又称为并行ADC）使用一排比较器来将输入电压和一系列的参考电压进行比较，因此从输入的模拟电压到数字量的转换速度非常快。存在的问题是高速ADC往往比其他类型的ADC要昂贵，并且由于其复杂性，分辨率通常比其他类型的ADC要低。

将一个模拟信号转换为数字量的过程称为采样或量化。ADC有两个主要参数：采样率和分辨率。采样率是每秒钟采样的个数（samples per second，SPS），即模拟输入信号被转换为数字码的频率。ADC的采样频率越高，其价格也会越高。分辨率决定了每次采样的精确度。比如，8位的ADC返回一个8位码来代表被采样的输入信号，这意味着输入信号被量化为256个离散值中的一个，而12位的ADC会将输入信号量化为4096个值中的一个，这样就得到了更精确的结果。然而，ADC的分辨率越高，其价格也就越高。此外，并不是所有时候都需要高分辨率。例如，如果你在对一个温度传感器进行采样，该

传感器的输出范围是0℃~100℃，精确到±0.5℃，那么这个传感器只有200个有意义的电压级。对这个传感器来说，用一个8位的ADC就足够了。如果你用一个12位的ADC来对它进行采样，增加的分辨率将是多余的。

ADC将输入信号转换为一个数值，该数值代表输入信号与给定基准电压的比值。例如，如果ADC的基准电压为5V，输入电压为3V，则输入电压与基准电压的比值为60%。所以，对于一个用255代表最大值的8位ADC来说，这个输入电压转换后的结果将会是153（0x99）。从你的角度看，从ADC得到一个数153，反过来也可以根据这个数计算最初输入的模拟电压：

```
输入电压 = (采样值 / 最大值) * 基准电压
        = (153 / 255) * 5
        = 3V
```

采样频率

采样频率对信号的量化结果有非常大的影响，因而也影响到了软件对转换结果的处理。图13-6所示的例子中，对一个正弦信号进行采样，使用的采样频率等于信号的正弦周期，采样点在信号的最高点。采样过程中信号在改变，而我们选择的采样频率使得每次得到的值都相同，所以我们得到的是对真实信号完全错误的描述。对于采样软件来说，每次采样都返回同样的值，所以最终显示给我们的信号将是一条水平直线！

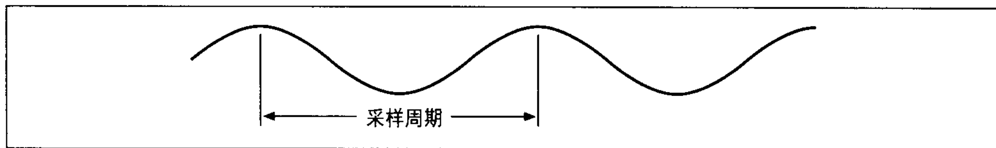


图13-6：采样频率选择不当导致信号读取不准确

如果选择的采样频率是信号最高频率的两倍（或更多），我们将会得到信号更多的细节（如图13-7所示）。这个采样频率被称作奈奎斯特频率（Nyquist frequency），它是我们能得到可用采样结果的采样频率下限。如果采样频率低于奈奎斯特频率，采样结果将会是一些错误的信号（比如前面所述的将一个正弦信号显示为一条直线），这种现象被称为失真。

注意： 你可能曾经在一部旧西部影片里看到过一辆四轮马车，即使是正在向前走，但其车轮却是向后转的，这就是失真的一个实例。摄像机的帧率（frame rate）对车轮的转动进行了有效地采样，然而由于车轮转动的速度比帧率稍慢，在每一帧中车轮并没有完全转完一整圈，所以在连续的帧中，车轮看起来比在上一帧稍微向后倒了一点，这样车轮看起来就好像在向后转——这就是行为失真！

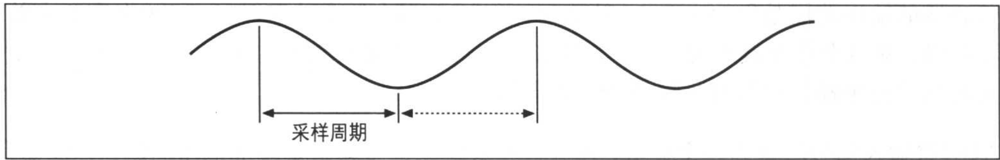


图 13-7：更短的采样周期

采样频率越高，采样结果就会越精确。由于采样是根据幅度（ADC 分辨率）和时间（采样频率）对信号进行量化，所以量化误差总是存在的（如图 13-8 所示）。

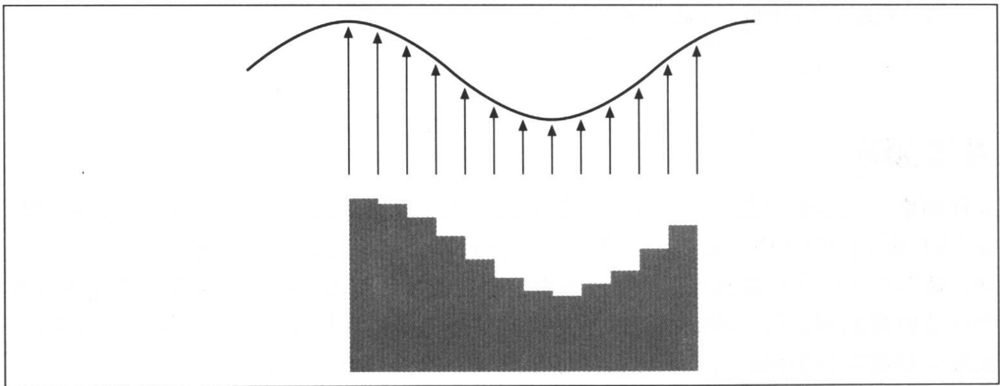


图 13-8：采样周期和相应的量化

可以看到，原始信号 —— 平滑的正弦波 —— 在采样后会成为一条锯齿状的曲线。如果你是在监测温度，这也许已经足够了，因为你可能不关心温度信号是如何变化的，你感兴趣的只是某些特定时间间隔上的有一定精度的温度值。在这种情况下，上面的量化结果不会造成什么问题。

不过，如果你是在对音频进行采样，这种量化效果就可能会带来问题。通过增加采样频率，可得到对原始信号一个更精确的表示（如图 13-9 所示）。

语音邮件系统的采样频率可能只有 8kHz，其分辨率可能为 12 位，因此合成后的声音质量就会受到限制。而 CD 音频使用的采样频率是 44.1kHz，采样结果用 16 位数据表示，其质量有很明显的提高。DVD 音频使用 44.8kHz 的采样频率，数据是 24 位，它的保真度更高。为了进一步提高声音质量，当数据被转换回模拟信号时，CD 和 DVD 播放机都有特殊的输出滤波器来对转换进行平滑。

最后的建议是：仔细选择 ADC 的分辨率和采样频率，精确地记住你要采集的对象是什么，以及要用它来做什么。

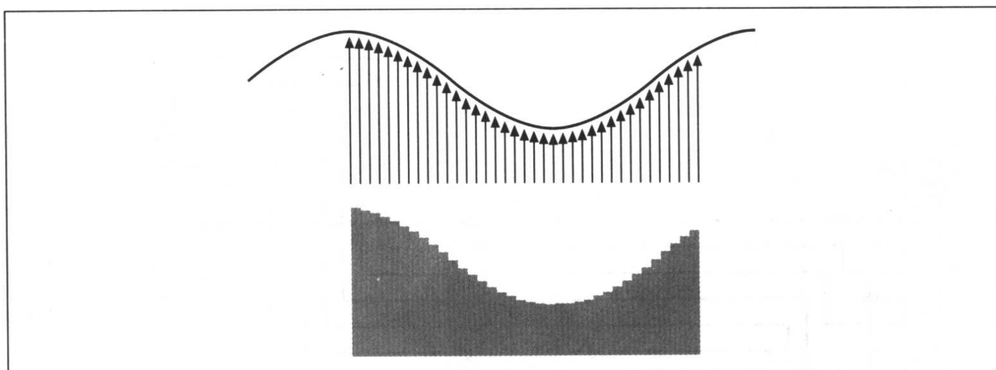


图 13-9：高采样频率量化误差小

连接外部 ADC

对于每个具体应用，有很多ADC都可以使用，从用于进行简单电压转换的低价、低速ADC到用于进行音频流采样的高速、高精度（也很昂贵）ADC 都可选择。许多微控制器都有内置的ADC子系统，这使得与模拟接口的连接变得很简单。然而，如果处理器没有包含ADC，或者它的 ADC 对于你的应用来说不适合，你就必须增加一个外部设备了。

一款不错的用于传感器领域的通用 ADC 是 Maxim MAX1245，它有 8 个模拟输入通道，每秒可进行 100000 次采样，分辨率为 12 位（类似部件的分辨率为 8~16 位，接口类型有 SPI、I²C 和处理器总线等）。MAX1245 有一个内部采样保持电路，避免变化的信号在转换的时候破坏转换结果。MAX1245 通过一个接口连接到主处理器，这个接口和 SPI、Microwire（微总线），以及 Texas Instruments 公司的 TMS320-series DSP 处理器中的串行接口（如图 13-10 所示）都兼容。可以看到，MAX1245 很容易使用。在图 13-10 中，模拟信号通过左边一个 16 引脚的连接器 —— IDC 头输入。注意连接器上的引脚每隔一个就接地，这意味着连接线中每隔一个就是地线，其目的是抑制噪声对模拟信号的干扰。

DOUT、**DI** 和 **SCLK** 信号分别对应于一个处理器 SPI 接口的 **MISO**、**MOSI** 和 **SCLK** 信号，$\overline{\textbf{CS}}$ 只是通过处理器的 I/O 线来产生。

通过 SPI 接口向 ADC 发送一个指令开始进行转换。开始指令只是一个字节，这个字节指定转换通道和这个特定转换所需的其他 ADC 设置（如需更多的软件接口信息，请参考 MAX1245数据表）。MAX1245使用一个内部的时钟源或外部提供的时钟源来对转换过程进行驱动。当使用内部时钟源模式时，输出即 **SSTRB**（Serial Strobe，串行通道）在转

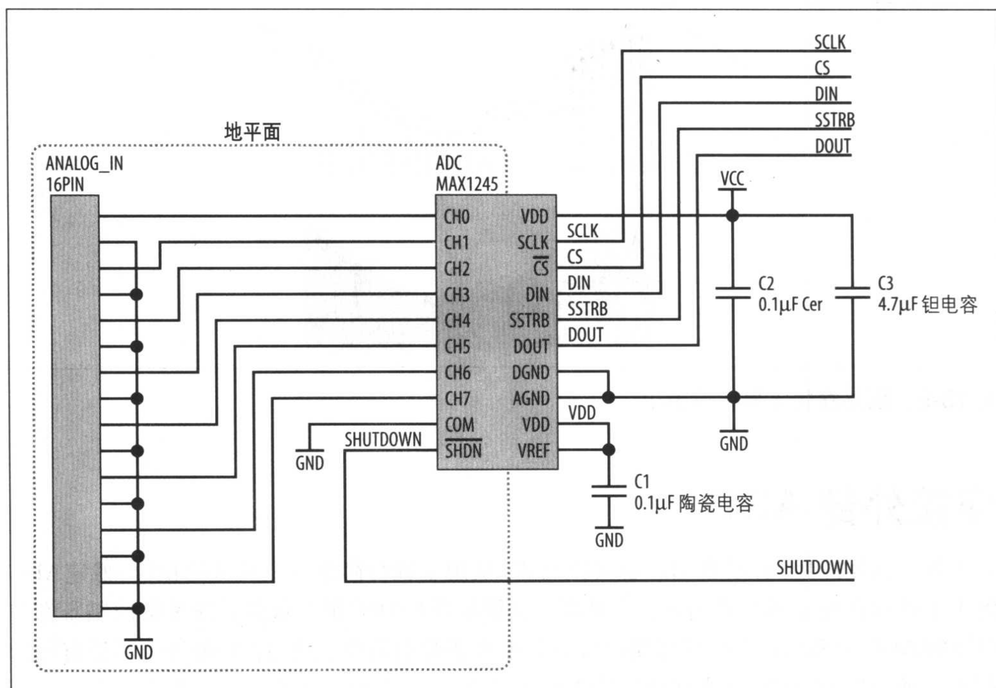


图 13-10：MAX1245 接口

换开始时变成低电平，转换一旦结束就变回高电平。使用外部时钟时，**SSTRB**会在首要数据位处理之前的时钟周期内产生一个上升沿脉冲。通过作为一个中断输入，**SSTRB**也可以用于向主处理器传达一个转换结束标志。换句话说，当使用外部时钟模式时，一旦开始指令发送出去，转换结果也就就绪了。

MAX1245 可以进入低功耗模式，这既可以通过硬件，也可以通过软件控制来完成。MAX1245 有一个输入引脚 **SHDN**，当这个引脚为低电平时，就将 ADC 置为低功耗工作模式。然而有趣的是，**SHDN** 还被用来指定 MAX1245 内部采样的时钟频率。将 **SHDN** 置为高电平，就将时钟频率设置为 1.5MHz；反之，将其悬空（不连接），就将时钟频率设置为 22.5kHz。如果 **SHDN** 是由一个微控制器的 I/O 引脚来驱动的，将引脚的状态从输出变为输入就可以将 **SHDN** 悬空，通过这种方式，即使引脚连接着，你仍然可以使用非连接功能！ MAX1245 还可以通过软件设置为低功耗模式。如果开始指令的最后两位都为 0，则 MAX1245 就被停止。通过软件来停止 MAX1245 的好处是，你只用一条指令就可以请求一个转换，然后将芯片停止。MAX1245 在停止前完成转换工作，不过其接口仍然处于活动状态，这样转换的结果就可以在时钟控制下送到微控制器。

MAX1245的电源（**VDD**）可以为2.7V~3.3V，它有3个接地引脚：**COM**、**DGND**和**AGND**。**COM**是模拟输入信号的接地，**DGND**是ADC的数字部分的接地，而**AGND**是ADC模拟部分的接地。这三个接地需要连接到同一个点上，该点要靠近**AGND**，称为星型接地点（star ground point）。两个电源输入引脚（VDD）需要两个退耦电容来消除电源电压的噪声。这两个退耦电容分别为0.1μF和4.7μF，并且应该尽可能地靠近星型接地点。对于噪声比较严重的电源，应该在供电电源和**VDD**之间串连一个10Ω的电阻。为防止对输入的模拟信号造成干扰，附近的数字信号都应该被屏蔽，另外还应该对MAX1245进行接地屏蔽来进一步将其和噪声隔离（更多关于噪声和屏蔽的信息参见第6章）。

前面已经讲述过如何向一个微控制器添加ADC，现在就让我们看一些具体的例子。下面要谈到是一些传感器以及如何将它们和ADC连接起来。传感器有很多种不同的类型，其生产厂商也有很多。有很多传感器难以使用，接口实现也不方便，并且实际付出的工作量比看起来所需的要多得多。但并不是所有的传感器都完全一样，我挑选出了其中易于使用，并且在进行设计时需要工作量极少的一部分。电子学可能很难，但它并非总是如此，不是吗？

温度传感器

我们先来介绍一个简单的应用——温度传感器。这个小传感器的应用范围很广，最显著的例子就是环境监控或气象站；不过你也可以拿它来感应室内的温度，并对加热或制冷系统进行适当的控制。将其和数据记录仪相连，你就拥有了一个温度记录器，这种装置用在水果、蔬菜、冷冻食物和鲜花的运输过程中，来确保这些物品以最好的状况出现在市场上。它还能用于运送血液和病理样本的过程中，来保证这些对环境温度要求极高的物质不受破坏。

美国Analog Devices公司的AD22100和AD22103温度传感器使用起来很容易。它们都是3引脚器件（注2），只需要电源（V_S）和地，就可以输出一个与温度成比例的电压（如图13-11所示）。AD22100要求一个5V的电源，而AD22103需要的电源是3.3V。

还有什么比这更简单的呢？

AD22100的温度测量范围是−50℃~+150℃，所以其输出电压相当于22.5mV/℃；AD22103的测量范围是0℃~100℃，其输出电压为28mV/℃。这两个传感器的传递函数（输入和输出的关系）由下面的式子给出：

注2：　这些器件也有8引脚贴片封装的芯片，只是其中的5个引脚没有使用。

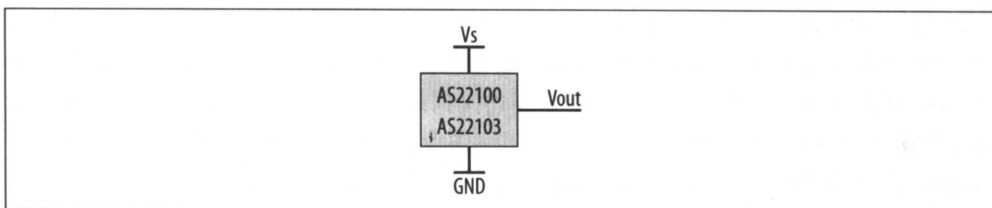


图 13-11：AD22100/AD22103

```
Vout = (VS / 5) × [1.375 + (0.0225 × TA)]          AD22100
Vout = (VS / 3.3) × [0.25 + (0.028 × TA)]          AD22103
```

这里的 V_{out} 是输出电压，V_S 是电源电压，T_A 是环境温度。

因此，将等式反过来，得到温度和输出电压之间的关系为：

```
TA = (((Vout × 5) / VS) - 1.375) / 0.0225          AD22100
TA = (((Vout × 3.3) / VS) - 0.25) / 0.028          AD22103
```

举例来说，如果使用一个 AD22100 温度传感器和电压为 5V 的电源（V_S = 5V），则上面的等式就会变得简单：

```
TA = (Vout - 1.375) / 0.0225
```

因此，如果我们测得一个输出电压为 1.94V，则相应的温度将是 25.1℃。

连接温度传感器和 ADC 很简单。传感器的输出可以直接连接到 ADC 的输入端。由于温度变化相对比较慢，我们还可以在传感器和 ADC 之间增加一个 RC 滤波器，来消除在输出端可能会出现的噪声，如图 13-12 所示。

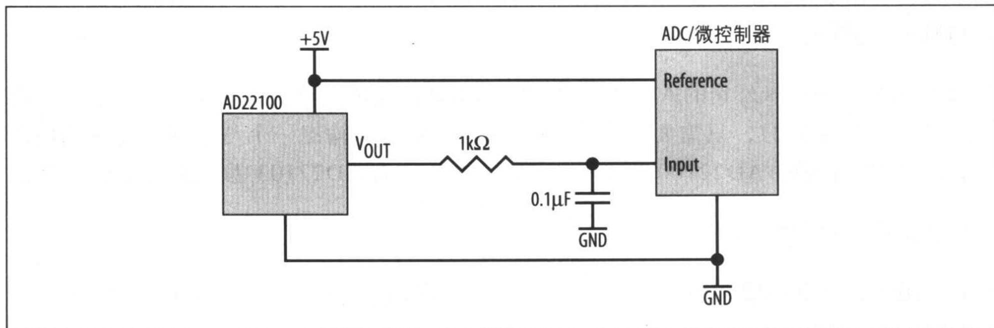


图 13-12：带有 RC 滤波器的 AD22100/AD22103

光电传感器

现在我们来看一看光电传感器。这类传感器最常见的用途是监控自然光的强度，还可以使用这个结果来控制人工照明系统。你还可以将这个传感器和一个有向性光源（如通过光栅的发光LED）结合在一起，就可以得到一个安全探测器。只要这个传感器可以“看到”LED，就表示一切正常。一旦光被挡住，你就可以知道有人或别的物体从传感器和LED之间经过。

注意： 我的公司使用一种特别的传感器，在一个小的数据记录仪上你将会看到这种传感器。我们有一个客户是研究南海信天翁（一种大海鸟）的生物学家，这个研究是正在进行的保护鸟类计划的一部分。信天翁一次可以飞行几年，在海风中环游世界。数据记录仪极小（比你的小指还要小），重量只有几克，用来绑到信天翁的腿上（数据记录仪经过精心设计，确保在任何情况下都不会对鸟造成不良的影响）。那个小传感器就用来记录信天翁所经之处的阳光强度。

通过比较所记录的日出日落和数据记录仪上的参照时钟，再观察曙光持续的时间，就可以计算出该地方的经度和纬度。这样，对光强的简单记录就可以用来跟踪信天翁环游世界时的飞行路线。

根据记录中光的分布图，还可以得到关于信天翁的一些行为的信息。你可以判断出它在飞行时脚是缩进羽毛里还是悬空的，是否停在水面上休息，因为每种行为都有一个唯一的光分布与之相对应。你还可以看到月亮在夜间光（强）中留下的痕迹，以及哪天是阴天哪天是晴天。当这只鸟夜间经过一个单独的亮光源的时候，传感器甚至可以探测出带有超导干涉装置的船。一个简易的传感器可以告诉你非常多的信息。更多信息参见 *http://www.oreillynet.com/pub/a/oreilly/cprog/news/albatross_1202.html*。

另外还有许多商业用途的光电传感器，我们接下来将看一看TAOS TSL250R传感器。TAOS（*http://www.taosinc.com*）即Texas Advanced Optical Solutions公司，是久负盛名的德州仪器公司的一个新生子公司。TSL250R（如图13-13所示）由一个光电二极管（一种能对光作出反应的半导体）和一个集成放大器组成。就像我们前面说的温度传感器一样，TSL250R只需要电源和接地，就可以输出一个和光强成比例的模拟电压。图13-14给出了TSL250R的光谱响应，其响应范围可以从紫外线（左边）到红外线（右边），在可见光部分响应达到最大。

TSL250R的输入电压为2.7V~5.5V，电流通常只有1.1mA，其基本电路非常简单（如图13-15所示）。

当输入电压为5V时，这个传感器的最大输出电压（在最大光辐射下）不会超过4V。因此，不需要附加的放大电路，就可以直接将传感器接到一个（参考电压为5V）ADC的

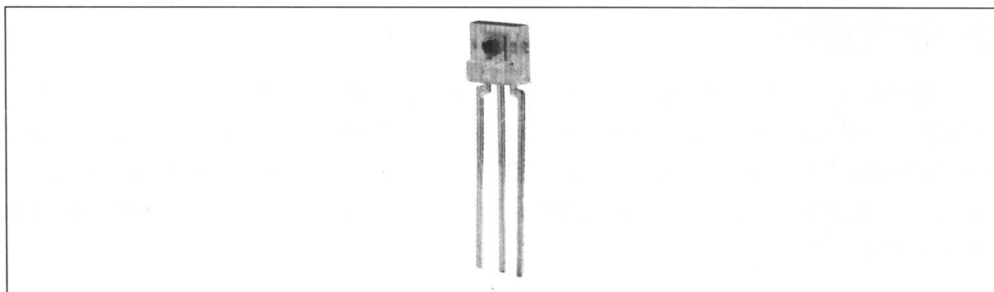


图 13-13：TAOS TSL250R 光电传感器

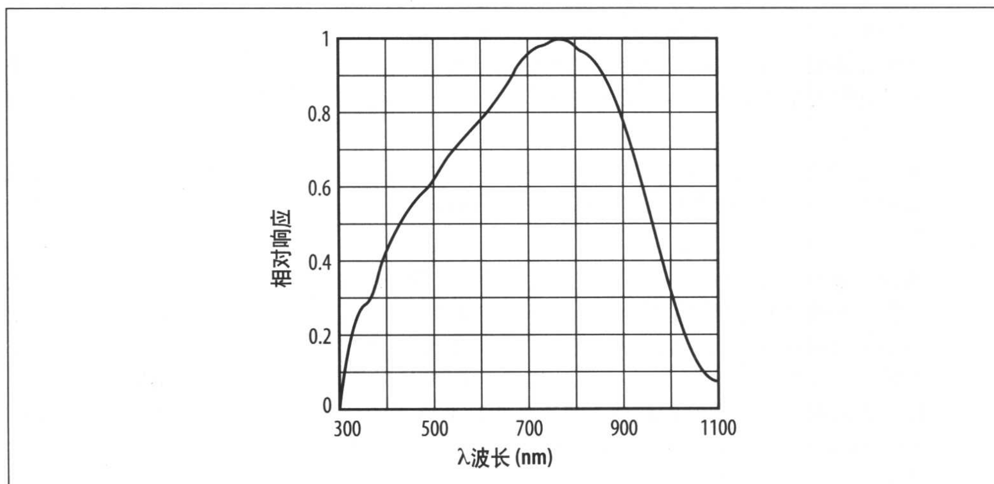


图 13-14：TAOS TSL250R 的光谱响应

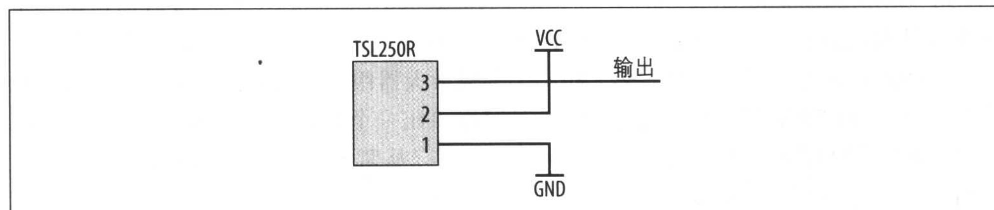


图 13-15：TAOS TSL250R 的用法

输入端。在这里，由于传感器的输出范围小于 ADC 的输入范围，所以会损失一小部分的分辨率。对于 8 位的 ADC，一个 4V 输入对应于 `0xCC`，因此这个传感器输出的范围就是 `0x00~0xCC`。根据具体应用的不同，这可能不会造成什么问题。例如，当你感兴

趣的只是对明暗差异的检测，或者给定的低光阈值在输出范围之内时，传感器完全可以胜任这个工作。

放大光电传感器

如果你想对传感器的整个输出范围进行采样，就需要对其输出进行放大。由于传感器的最大输出是 4V，而 ADC 的参考电压是 5V，所以放大器的增益必须是 1.25。

一款比较好的通用OP放大器是Analog Devices公司的AD623，它具有rail-to-rail功能，能够在单电源下工作，要求的电流很小，并且非常容易使用。Analog Devices公司早已做了许多出色的工作，使得AD623仅仅需要一个外部电阻就可以对增益进行设置。电阻值用下式计算：

```
RG = 100kΩ / (Gain - 1)
```

因此，当我们要求增益为 1.25 时，需要的电阻是：

```
RG = 100kΩ / (1.25 - 1)
   = 100kΩ / 0.25
   = 4 KΩ
```

这个电阻的偏差（精度）应该为1%或更好。市场供应的标准电阻误差通常为5%，不能满足精度要求。

图 13-16 所示是一个光电传感器 TSL250R 和 OP 放大器 AD623 相连接的电路。

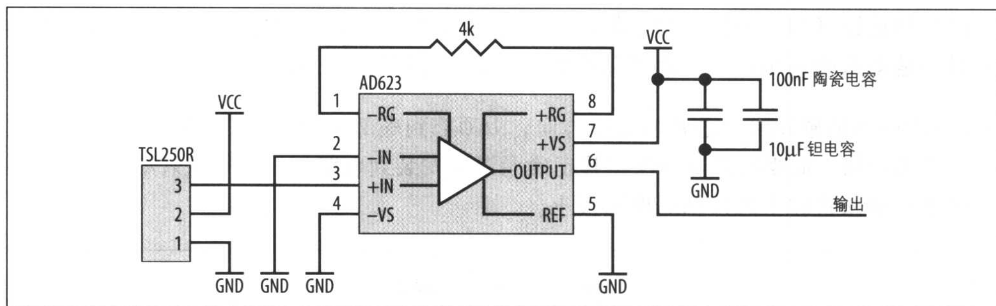


图 13-16：对光电传感器 TSL250R 的输出进行放大

AD623 的同相输入端（引脚 3）连接到 TSL250R 传感器的输出端（引脚 3），而它的反相输入端接地。增益电阻连在引脚 1 和引脚 8 之间，负电源 **–VS** 连接到地来实现单电源工作，而正电源 **+VS** 连接到 **VCC**，并且和地之间通过两个电容相连来实现退耦。这个

OP放大器的参考输入端（REF）也和地相连，其输出（引脚6）和ADC的模拟信号输入端直接相连。

加速计

现在，来看一类非常有趣的传感器。Analog Devices公司确实开发了一些非常好的加速计，下面来探讨如何将加速计ADXL150连到一个嵌入式系统上去。加速计可以用在很多地方，而不只是用来测量车辆的线性加速度。ADXL150是一个单轴（一维）加速计，其分辨率为10mg，最大测量范围是±50g。对于双轴（二维）传感，则应选择ADXL250。

注意： g是加速度单位，1g大约等于9.8米/秒^2。当乘坐喷气式飞机的时候，在飞机转弯时你将会受到大约2g的压力。一架战斗机急速转弯时里面的人受到的压力将会达到8g，如果没有特殊的装备，战斗机的飞行员在8g的压力下会丧命。因此，测量范围为±50g的ADXL150可以测量相当大的压力。

10mg这么小的分辨率意味着这个传感器可以用来测量轻微的振动和移动，在地球物理应用中可以将它用在一个地震检波器中，或者在矿山、隧道或建筑物等场所中测量地面的振动或移动。将3个加速器在空间相互垂直放置来对运动进行检测，就可以得到一个精确的3D运动记录，这个装置还可以作为一个数字化的水平仪，来感应地球重力场的方向。还可以用它来检测猛烈的物理冲击，比如碰撞试验的测量。想看看，是不是搬家工人在搬家的时候把出自Granny的精美水晶玻璃器皿打碎了？只需在包装盒里放一个这样的加速计（和一个适当的小数据采集系统），你就可以看到搬家公司的工人在搬运的时候是多么的不小心。由此可以看出，这块芯片可以有很多应用。

ADXL150的敏感轴沿着芯片的长度方向，由顶部到底部（如图13-17所示）。在使用这个器件的时候，重要的是要确保将其正确无误地安装到电路板上，不能只用焊锡，还要用强力胶水来将这个芯片粘到电路板上。

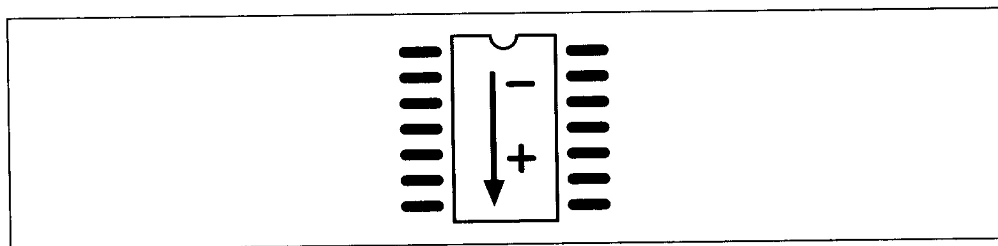


图13-17：敏感轴

ADXL150不需要外部元件（除了为电源退耦），它是一个完全独立的单元，不但内置了传感器，而且还内置了信号调节器和放大器，其输出可以直接接到ADC上。使用ADXL150的电路原理图如图13-18所示。

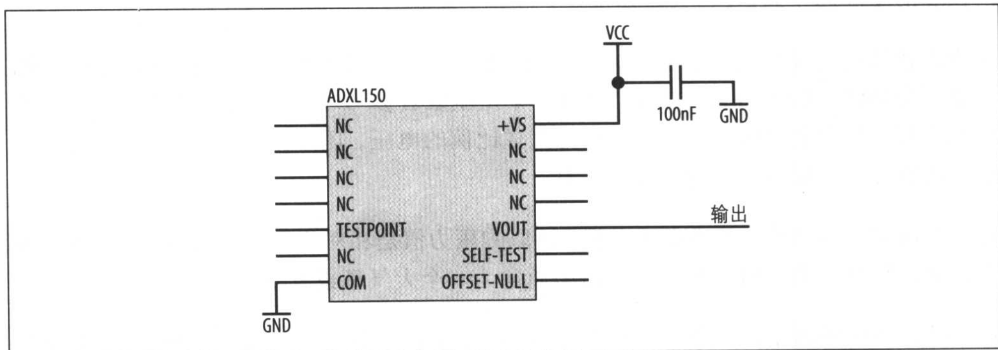


图13-18：使用ADXL150

大多数的引脚都是不连接的（No Connection，NC），可以忽略，如TESTPOINT引脚和SELF-TEST引脚。TESTPOINT引脚只在生产的时候使用，之后就不再使用。

ADXL150的电源电压要求在4V~6V之间。不过，为了达到最理想的情况，应该精确到5V，越接近5V，测量到的加速度结果就越准确。输出电压既与加速度成比例，也与电源（V_S）电压成比例，其关系由下面的式子给出：

V_{OUT} = VS/2 - （灵敏度 * VS/5 * 加速度）

灵敏度的值随着器件的不同而不同，其范围为33.0~43.0，额定值为38.0。标准的灵敏度值对应的测量范围是±50g；然而当将传感器的输出和OFFSET-NULL引脚连接起来的时候，灵敏度会加倍（假定测量范围为±25g）。

SELF-TEST引脚既用来验证传感器内部机构是否正常工作，还用来对信号调节和放大部分的电路进行验证。从这个引脚人为地输入一个逻辑1，就在传感器上施加了一个力，这样就可以验证传感器是否能正常工作。

压力传感器

接下来了解一下压力传感器。这种传感器最常见的应用是在天气的监测和预报中用来测量大气压。不过压力传感器还可以用在其他许多方面：用在车辆里测量各种压力，用在洗衣机里测量水位，用在生物医学应用里测量血压。由于大气压随着海拔高度的变化而变化，压力传感器还被用来测量高度，同样也可以测量海洋深度。

使用压力传感器的时候，要测量的物质对器件的影响很大。记住这些都是敏感的电子元件，流体和腐蚀性气体会对它们造成毁坏。因此，如果测量的不是清洁、干燥的气体，就需要对传感器有一定的环境保护措施。具体措施视应用、要防止的环境条件，以及经济预算的不同而不同。

压力传感器是通过测量一个膜片的变形来工作的，这个膜片将一个空腔分隔为两个室。一个室和要测量的压力直接相通，另外一个室里装着参考气压。两个室中的压力差导致膜片变形，形变量被转换成一个和压力差成比例的电压。压力传感器有三种类型：绝对压力传感器、差动压力传感器和气压计。

在绝对压力传感器中，参考室被封闭。读取的压力值是相对于一个绝对压力而言的，故有此名。绝对压力传感器参考室的压力通常为一个大气压或真空。

在差动压力传感器中，参考室没有被封闭，而是使用一个外部的压力作为参考。这种传感器用来测量两部分气体或液体之间的相对压力。当参考室密封参考压力恒定时，差动压力传感器可以作为绝对压力传感器来使用。

气压计是由差动压力传感器变化而来的变种，是将后者的参考室直接和大气相通得到的。因此，它测量的压力是以大气压作为参考压力的，需要考虑不同大气压（比如由于气候或高度引起的不同）的情况。气压计一个很有趣的应用是测量航速。如果测量室直接面对流动的气体（如由航天器的电机引起），而参考室和大气相通（要避免流动气体的影响），则两个室的压力差可以用来计算航天器的航速。

了解了前面的内容之后，现在让我们来看一些压力传感器。第一个传感器是 Freescale（前身为 Motorola）公司的 MPXA6115A 绝对压力传感器（如图 13-19 所示）。它的工作电压是 5V，输出电压为 0.2V~4.8V，输出电压与压力成比例，测量范围为 15 千帕~115 千帕（帕，指帕斯卡，是一个压力单位）。大多数压力传感器都需要外部的信号调节、温度补偿和信号放大，但 MPXA6115A 不同，它将这些功能需要的部件都集成在一小块部件里面。MPXA6115A 有 8 个引脚，有些引脚有通气管，有些没有。

NC 引脚不需要连接，所以应该保留不连。仅仅需要在电源处增加一个退耦电容，在输出处并联一个电阻和一个电容。其输出可以直接接到一个 ADC 的输入端。

我们要看的第二个压力传感器还是一个绝对压力传感器。但不同的是，这个传感器包含一个内置的 ADC，输出的不是模拟信号。它连接到微控制器的 SPI 接口。它直接输入数字信号，因此对噪声和干扰不敏感。这个压力传感器叫作 KP100，由德国慕尼黑的 Infineon Technologies 公司（*http://www.infineon.com*）生产。

图 13-20 所示是一个基于 KP100 的电路原理图。

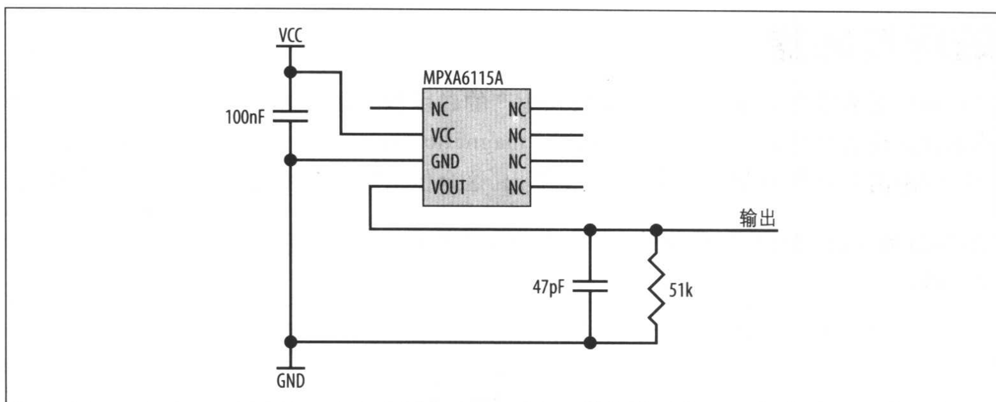


图 13-19：Freescale MPXA6115A 压力传感器的连接

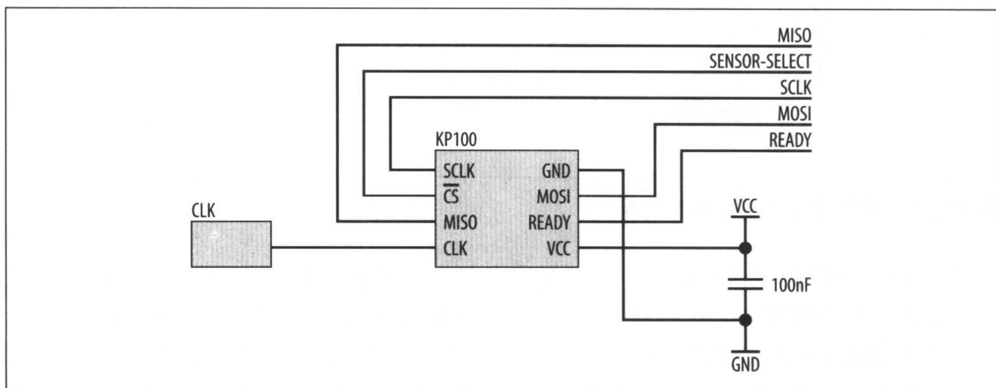


图 13-20：K100 压力传感器电路

这个传感器工作的电源电压是5V，电源通过一个100pF的电容接地来退耦，以减少噪声的干扰。KP100有个标准的SPI类型的接口，可以像连接到任何SPI设备那样连接到微控制器。它有一个 **READY** 输出端，可以用来中断主处理器，也可以只是接到一个备用的I/O端口，作为一个数字状态标记。KP100还需要一个单独的时钟输入（**CLK**），时钟频率可以是4MHz，也可以是8MHz。如果处理器的运行频率也正好为4MHz或8MHz，那么传感器就可以使用处理器的时钟输入。不过，即使处理器的时钟频率不同，KP100的时钟也很容易用一个时钟模块产生。这些4引脚的时钟模块可以提供很多标准频率，只需要电源和接地就可以产生时钟输出。

磁场传感器

最后我们要看的是 Analog Devices 公司生产的磁场传感器 AD22151。这种传感器首要的用途是位置和距离探测。用一个磁源（magnetic source）作为参考点，传感器和磁源之间的距离可以由测得的磁力很容易地算出。这个传感器有内置的温度补偿和放大模块。

图13-21所示是AD22151的电路。这个电路比我们前面看过的其他传感器电路要稍微复杂一些。

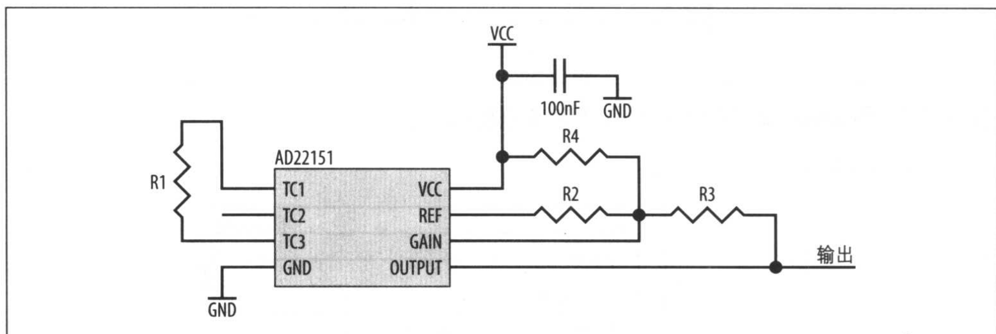


图 13-21：AD22151 磁场传感器电路

传感器 AD22151 的工作电压为 5V，电源和接地之间连接一个 100nF 的电容来退耦。该传感器需要四个电阻来对传感器进行校正。电阻 R1 是温度补偿电阻，针对不同的磁场，应该连到引脚 1 和 3 或引脚 2 和 3 之间。对于比较大的外部磁场，R1 连到引脚 1 和 3 之间，就像图 13-20 中那样；对于小磁场，连接到引脚 2 和 3 之间。AD22151 数据表中有 R1 和补偿量之间的对应曲线。核对你所使用磁源（注 3）的生产商提供的需要补偿的温度值，根据该值查找数据表来确定 R1 的大小。

要想得到相关信息，可以试试以下链接：*http://www.magtech.com.hk*、*http://www.east-industries.net* 或者 *http://www.millennium.com.hk*。

电阻 R2 和 R3 用来设置内部放大器的信号增益，R4 提供一个电压偏移。传感器的技术手册中包含有根据特定的需求来计算这些电阻值的等式和相关数据。

这个传感器电路的输出可以直接连到一个 ADC 的输入端，来供后者进行采样。

注 3： 在此假定你正在使用专门针对这类应用并且数据可用的磁体，而不是你从别的什么地方找到的一块磁体。

数/模转换

到目前为止，我们已经了解了如何检测现实世界中的各种物理量，并将它们转换为数字数据。现在让我们看看反过来该怎么做：把数字数据用一种叫作数模转换器（DAC）的芯片转换为模拟信号。我们还将讲述如何只用一根数字 I/O 线输出模拟信号。

所有的 DAC 都有一个数字输入端（微控制器总线、SPI 总线或者 I²C 总线），而它们有一个或多个模拟输出通道。

Maxim 公司的 MAX525 是一个 11 位的 DAC，它可以使用 SPI 总线连接到主处理器。这个 DAC 有四个模拟输出通道，芯片本身集成了两个输出放大器。每个放大器的反相输入端都是可以调节的，这样你就可以对放大器的增益分别进行改变。图 13-22 所示是 MAX525 的一个实例电路。

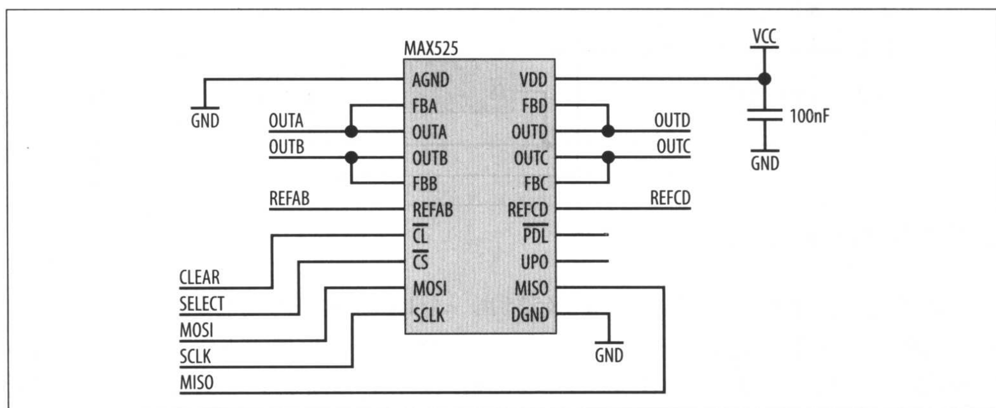


图 13-22：MAX525 电路

四个模拟输出通道是 **OUTA**、**OUTB**、**OUTC** 和 **OUTD**。这些引脚都直接连到它们各自的反馈输入端（**FBA**、**FBB**、**FBC** 和 **FBD**），以进行标准的单极工作。有两个参考电压输入：**REFAB**（供通道 A 和 B 使用）和 **REFCD**（供通道 C 和 D 使用）。这两个参考电压输入在任何时候必须比 VCC 低 1.4V，或者更低。每个通道的输出电压由下式给出：

```
VOUT = (VREF * 输入数字 / 4096) * gain
```

这里的输入数字指向该通道写入的数字值。在实例电路里面，增益是 1（关于如何设置增益，参考前面同相放大器部分内容）。如果参考电压为3.6V，数字值为`4095`（`0xFFF`），则产生的输出电压为：

```
VOUT   = (VREF * 4095 / 4096) * gain
```

```
            = 3.6 * 0.9997 * 1
            = 3.59V
```

与此类似，如果数字值为 2048（0x800），则产生的输出电压为：

```
VOUT= (VREF * 2048 / 4096) * gain
     = 3.6 * 0.5 * 1
     = 1.8V
```

注意，在电路图里，模拟地和数字地分开了。实际上它们应该连到一起，但只能连在靠近 DAC 的一个点上。

MAX525 有标准的 SPI 接口，可以连接到微处理器。多个 MAX525 可以按照菊花链形式连接到一起，以提高效率（如图 13-23 所示）。

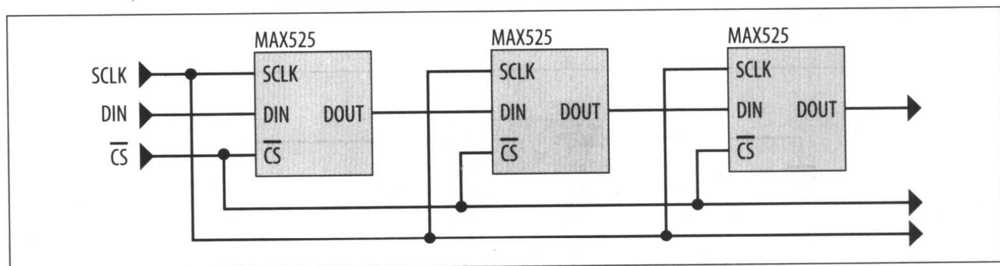


图 13-23：菊花链形式连接的 MAX525

MAX525 还有一个 **CL** 输入端，当 MAX525 由一个 I/O 数据线的低电平驱动时，这个输入端就向所有的输出端发送它们的最小值。这个 DAC 在软件控制下还可以设置为低功耗模式。**PDL** 输入可以将 MAX525 的低功耗模式禁止，当向这个 **PDL** 输入端输入低电平时，就不再允许将 MAX525 置为低功耗模式。在用输出来驱动一个关键电路或系统时，这点很重要，因为你不希望控制电压意外消失。最后要说的是，考虑周到的 Maxim 公司还提供了一个信号叫作 UPO（User Programmable Output，用户可编程输出）。这是一个通用输出，可以在软件环境下置为高电平或低电平，你可以根据需要来使用它。

现在，如果你想得到一个不是 1（非同式增益）的增益，那么输出放大器就需要外部电阻。图 13-24 所示是实现这个目的（只针对一个输出通道）的电路图。

在前面，我们就知道一个同相放大器的增益由下式给出：

```
Gain = 1 + R2 / R1
```

对于给定通道上的双极输出，可以用一个外部的放大器（有双极电源）来实现（如图 13-25 所示）。

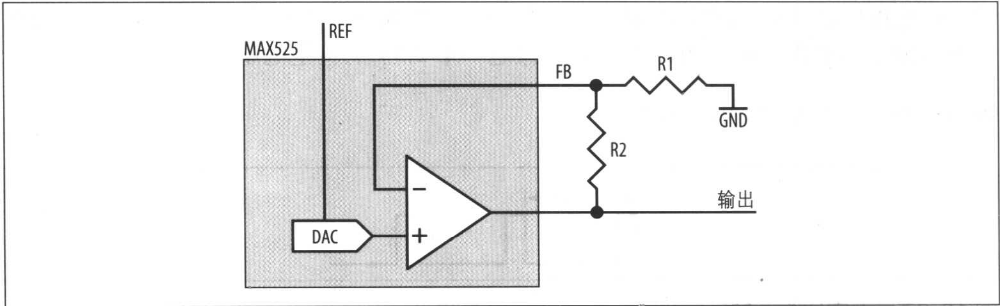


图 13-24：非同式增益的反馈电阻

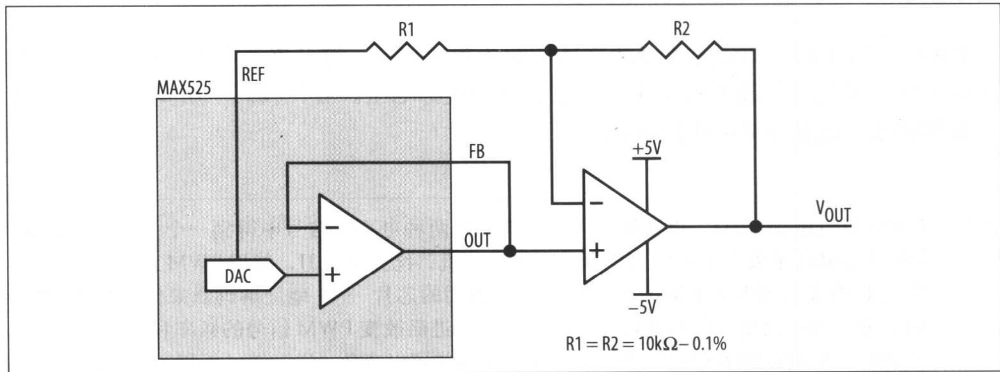


图 13-25：双极输出

脉宽调制

使用 DAC 来产生模拟输出电压是很显然的方式，但是还有一种方式，可以只使用一个设定为输出的数字 I/O 线来产生。这种技术被称作脉冲宽度调制（PWM）。

图 13-26 中所示为一个普通的方波。

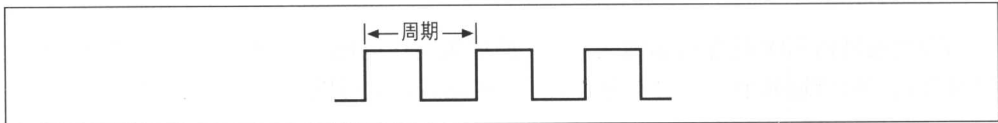


图 13-26：一个普通的方波

高电平和低电平的宽度相等，因此称这种波形的占空比（duty cycle）为 50%。换句话

说，就是其高电平正好占整个周期的一半。现在，假设这个方波的幅度是5V，那么它在一个周期上的平均电压就为2.5V，就好像是一个2.5V的恒定电压。

现在再看一下图13-27中的方波。

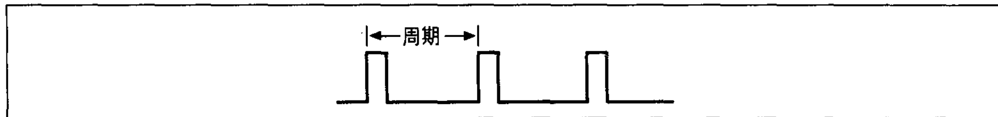


图13-27：占空比为10%

这个波形的占空比为10%，也就是说一个周期上的平均电压是0.5V。

在PWM的输出端，低通滤波器会将脉冲变为模拟电压，这个电压和PWM信号的占空比成比例。我们通过改变占空比来改变输出的模拟电压。看！我们不用DAC就可以实现数模转换，这就是PWM的初衷。

注意： PWM可以驱动LED，用于从本来只有高和低两种电平的信号中得到一个变化的光强度。PWM还可以用来产生音频。早期的台式计算机，比如苹果II，使用PWM来驱动扬声器。苹果II的设计者Steve Wozniak选用一个备用的芯片——地址解码器来作为他的PWM的信号。通过改变一个特定地址的访问频率，进而改变PWM信号的频率和占空比，因此就可以产生简单的音频，这种音频的音量和节拍可以变化。通过地址解码器就可以发声！

电机控制

使用嵌入式系统，你可以做的很有趣的一件事就是用它来精确控制一个物体的移动，不管这个物体是一个外部系统还是嵌入式系统本身。运动就意味着要有电机，在这部分你将会了解如何将一个嵌入式计算机和一个电机相连接。电机可以用于从在模型铁轨上控制火车到机器人试验以及介于二者之间的任何应用。然而，有一点需要提醒：如果你的软件和硬件是用来控制一个物体运动，一个故障就可能造成物理损坏，因此要格外小心。

我们假定电机由12V的电压来供电。12V的电源会让电机全速转动，同样地，6V的电压就会让它的转速减半。通过改变电机的供电电压，就可以改变它的转动速度。

用来驱动电机的电压可以通过几种方式产生，最常见的是用DAC来产生一个模拟输出电压，然后通过放大器将输出信号的电流和电压提升到电机转动所需的高度，电机的转速和放大器的输出电压成比例。但是，这种技术有个主要缺点，就是在很低功耗工作的时候，要求的输出电压可能会太低而不能真的让电机转动起来。

一个更好的方法是用PWM。图13-28所示是一个幅度为12V的PWM信号。

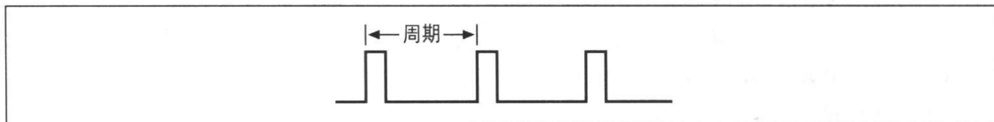


图13-28：占空比为10%的PWM信号

当占空比为10%时，这个PWM信号的有效模拟输出电压为1.2V，这个电压还不足以让电机转动起来。不过，我们不是用1.2V这个电压，我们实际上用的是12V，即PWM信号的最大电压来驱动电机。脉冲低电平的时候，不能够让电机转动起来，但是最大幅度12V就一定可以让电机转动，这就是PWM的好处。要改变转速时，只需改变脉冲的宽度而不是幅度。

用PWM可以让电机保持很低的转速，并能进行很好的控制。如果频率太低的话，脉冲会使电机转动不平稳，但选择一个较高的频率，就可以解决这个问题。

许多微控制器内部都有可编程PWM模块，可以很容易地产生PWM信号。即使处理器没有PWM模块，只需一根数字输出线，仍然可以在软件控制下产生PWM信号。

接下来我们看看使用PWM的电机如何与处理器连接。由于电机对电压和电流有要求，所以你不能期望将电机直接连到处理器的引脚上就可以让它转起来，还需要一个接口电路来获得PWM的输出，再用该电路将PWM信号转换成更高的电压和更大的电流。

图13-29所示是用来驱动小电机的接口电路的总体模型（比较粗略并经过简化），这种电路就是众所周知的H-bridge电路。

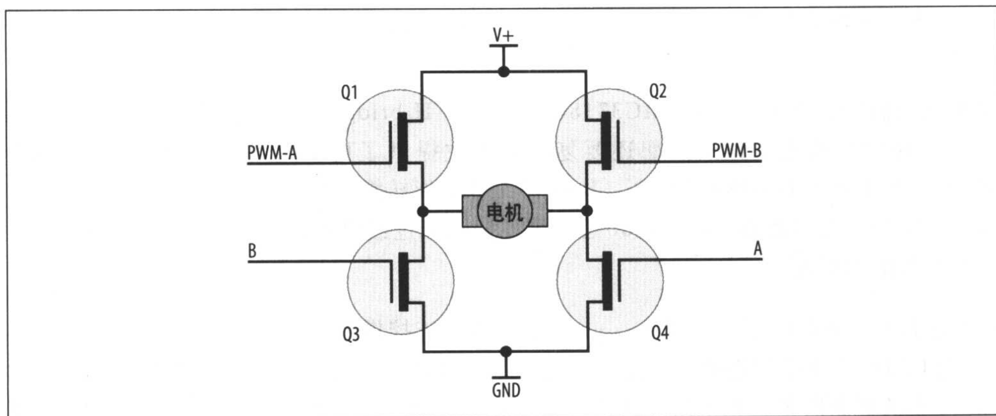


图13-29：用于电机驱动的H-bridge电路

它实际上没有像初看上去那样混乱。不必在意电路中的晶体管，它们只不过充当开关的角色。电机由 **V+** 供电，**V+** 供电后电机正向运转，如果将电源极性反过来的话，电机的转动方向也会改变。要驱动这个电路，需要四个处理器的输出来完成：两个 PWM（称为 **PWM-A** 和 **PWM-B**）和两根通用 I/O 线（称作 **A** 和 **B**）。最初，所有输出都是低电平，一切都是关闭的，电机不转动。

如果向 A 发送一个高电平，晶体管 Q4 就导通，将电机的右端接地。如果再向 **PWM-A** 发送高电平，晶体管 Q1 就导通，电机的左端就连接到了 **V+**，这样电机就开始转动。通过在 **PWM-A** 上产生一个 PWM 信号，可以控制电机的某方向的转速。

反过来，让 **A** 和 **PWM-A** 保持低电平，将 **B** 和 **PWM-B** 设为高电平，晶体管 Q2 和 Q3 就导通，电机就会向相反方向旋转。通过在 **PWM-B** 上产生一个 PWM 信号，可以控制电机在相反方向上的转速。

使用软件时要小心。如果 Q1 和 Q3 或者 Q2 和 Q4 都导通了，那就相当于把 **V+** 接地，二者之间的电阻很小！造成的结果是无法预料的！一个严格的H-bridge电路通常会避免这种情况的发生。

H-bridge 电路实际实现起来要稍微复杂一些，需要增加一些元件，比如保护二极管等。虽然可以用分离元件来设计一个这样的 H-bridge 电路，但是现在有一个简单的方法。许多生产商，比如 Freescale、International Rectifier 公司（*http://www.irf.com*）、M.S. Kennedy 公司（*http://www.mskennedy.com*）和其他的芯片生产商都在使用便捷的集成电路里面提供了 H-bridge 电路。

注意：如果你曾经在一个元件生产商的网站上寻找能在高电压下转换大电流的器件，可能会很失望，那么就到他们的“汽车元件”区去看看，因为这种器件有时会躲在那里。

现在让我们看一下 Freescale MC33186 —— 一个 H-bridge 电路的例子。这个芯片要比我前面介绍的桥式电路的思想还要复杂，但它提供了更多的功能，而且容易使用。MC33186 可以在 5V~28V 的电压（**V+**）下工作，能转换的电流高达 5A，它拥有逻辑输入，且与 TTL 逻辑兼容。MC33186 还内置了短路和过载电流保护电路，图 13-30 给出了这个芯片的电路图。

这个芯片有 3 个电源输入引脚 $\mathbf{V_{BAT}}$，它们都必须连接供电电压 **V+**。电源输入端要用一个 47μF 的电容连接到地来进行退耦。内部的充电泵（charge pump）也需要一个退耦电容，引脚 **CP** 提供了到充电泵的通路，它也需要通过一个 33nF 的电容。这个芯片还有 5 个接地引脚，同样地，这些引脚全部都要接地。

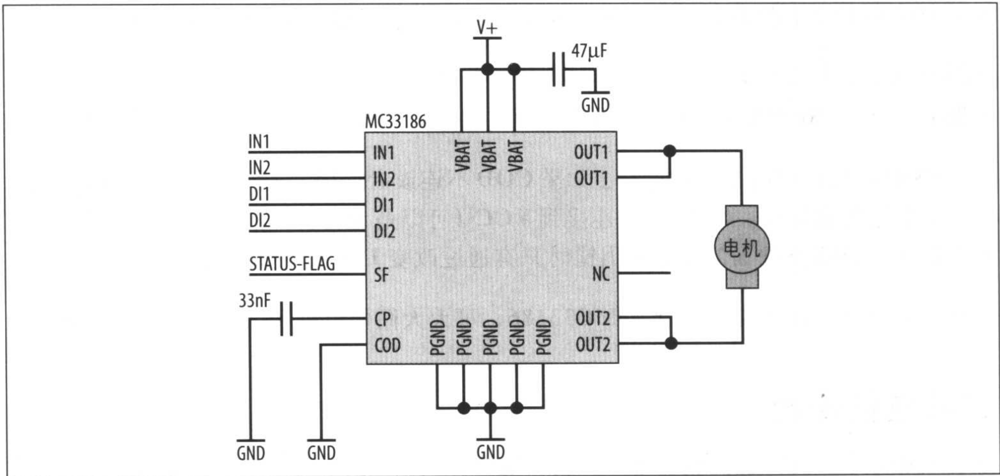


图 13-30：MC33186 电机驱动电路

OUT1 和 **OUT2** 两个引脚用来直接驱动电机，每个芯片都有两个这样的引脚，这样输出的大电流就不会只通过一个引脚流出了。

IN1 和 **IN2** 不但控制电机的速度，还控制它的旋转方向。**DI1** 和 **DI2** 用来使 MC33186 无效。这 4 个控制信号可以通过微控制器的 I/O 线来驱动。在正常操作情况下，**DI1** 是低电平，而 **DI2** 是高电平；**DI1** 变低电平或者 **DI2** 变高电平都会使 MC33186 无效，电机停止转动。表 13-1 给出了 **IN1**、**IN2**、**DI1** 和 **DI2** 是如何影响电机工作的。

表 13-1：MC33186 工作状态

DI1	DI2	IN1	IN2	OUT1	OUT2	电机
低电平	高电平	高电平	低电平	V+	地	正向旋转
低电平	高电平	低电平	高电平	地	V+	反向旋转
低电平	高电平	低电平	低电平	地	地	惯性滑动
低电平	高电平	高电平	高电平	V+	V+	惯性滑动
高电平	不管	不管	不管	高阻抗	高阻抗	无效
不管	低电平	不管	不管	高阻抗	高阻抗	无效

如果想要电机正向旋转，就向 **IN1** 施加一个 PWM 信号，而将 **IN2** 置为低电平。如果想使其反向旋转，就将 **IN1** 置为低电平，而向 **IN2** 施加一个 PWM 信号。PWM 信号的占空比决定电机的速度，实现起来非常简单。

如果 **IN1** 和 **IN2** 的状态相同，电机两端就没有电压差，也就不会驱动电机转动。

MC33186 的引脚 2（**SF**）是一个输出状态标记。当 MC33186 正常工作时，**SF** 为高电平，如果电机出错，SF 就变为低电平。因此 **SF** 可以用作一个中断，向主处理器报警电机出错。

输入 **COD** 决定出错时芯片的操作。如果 **COD** 不连或者接地，无论 **DI1** 还是 **DI2** 的变化都会对出错状态复位。如果 **COD** 连接到 **VCC**（可以是 5V 的电压，不一定非得是 V+），那么 **DI1** 和 **DI2** 就被置为无效。出错时只有通过改变 **IN1** 或 **IN2** 才可以复位。

使用集成的 H-bridge 电路，比如 MC33186，可大大简化电机和嵌入式系统的连接。

测量电机转速

由电机驱动的系统将会影响到电机的转动速度。如果电机带动一个大的负载，那么它的实际速度可能会比预定的要慢。在这种情况下，测量电机的实际转速很有用，这样嵌入式系统就可以根据情况进行相应的补偿。

测量电机速度最容易的方法是使用一个光学编码器模块，比如 Agilent 公司的 HEDS-9000 或类似器件。编码器由一个光源（LED）和一组光电探测器组成，这些探测器由一个称作编码轮的有槽盘片隔开（如图 13-31 所示）。这个盘片安装在旋转的电机的轴上，每次一个槽从 LED 和探测器之间通过时，探测器就会收到一个光脉冲，进而产生一个电子脉冲，脉冲产生的速度直接和电机的转速成比例。编码轮的分辨率是它的每圈槽数，或叫作 CPR（counts per revolution）值。HEDS 系列的编码器 CPR 值的范围为 96~2048。

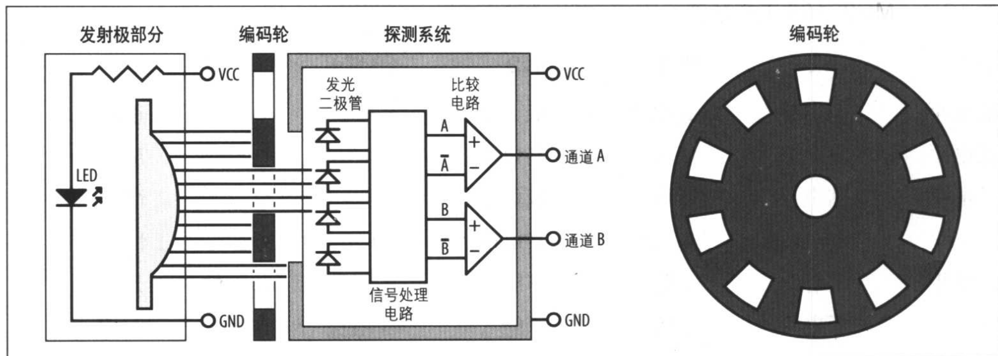


图 13-31：HEDS-9000 光学编码器和编码轮

HEDS-9000 光学编码器的工作电压为 5V，它有两个输出 **A** 和 **B**。这些输出来自两个相邻的光学传感器。如果编码轮朝一个方向旋转，**A** 就会在 **B** 之前被触发，如图 13-32 所示。

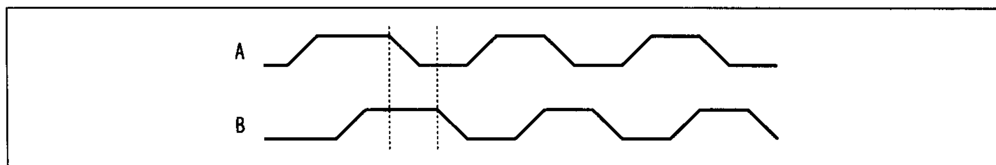


图 13-32：光学编码器的输出波形

如果编码轮是向相反方向旋转，则 **B** 将会在 **A** 之前触发（如图 13-33 所示）。

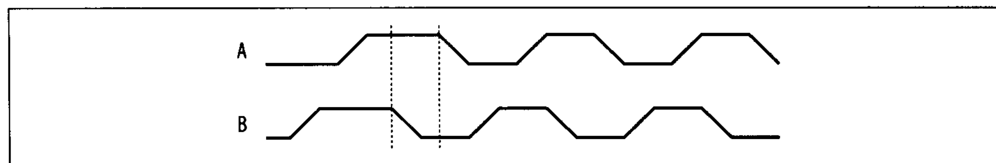


图 13-33：光学编码器的输出波形

脉冲到达的速率给出了电机的转速，而根据它们到达的顺序则可以判断转动的方向，这叫作正交编码。

大多数微控制器都有定时器 / 计数器输入端，它们可以测量外部的触发事件，比如这里谈到的。在软件控制下，可以用定时器来监测这些正交信号。不过，Agilent 公司开发了一系列称为正交计数器的器件，它们是 11 位的 HCTL-2000、16 位的 HCTL-2016 和 16 位可串联的 HCTL-2020。这些芯片都提供基于总线的接口，可以和处理器相连，并将正交信号转换成二进制数来表示电机的状态。一个16位的状态计数器能够在每个方向上测量 32767 个增量，这个数大概等于一个 CPR 值为 2048 的编码器 15 圈的槽数。要确定电机当前的转速和方向，处理器只需像读另外的存储器单元一样读取正交计数器就可以知道。正交计数器在输入端还有噪声滤波器，因而可以更可靠、更精确地确定电机的状态。

图 13-34 和图 13-35 所示分别为光学编码器和正交计数器的示意图。光学编码器被放置在一个单独的小电路板上，这样它就可以很容易地安装在电机轴旁。正交计数器在嵌入式计算机的电路板上。IDC 头（J1 和 J2）和一个带状电缆线将两个电路板连接起来。

正交计数器需要一个 14MHz 的时钟，这通过一个振荡器模块很容易实现。**CHA** 和 **CHB** 是来自编码器的正交输入，计数器有一个复位输入端 **RST**，它可以将计数器清零。要使 **RST** 有效就将计数器的值变为 0，意味着电机处在初始状态。**RST** 由微控制器的一个数字输出端来驱动，因而计数器的复位可由软件进行控制。

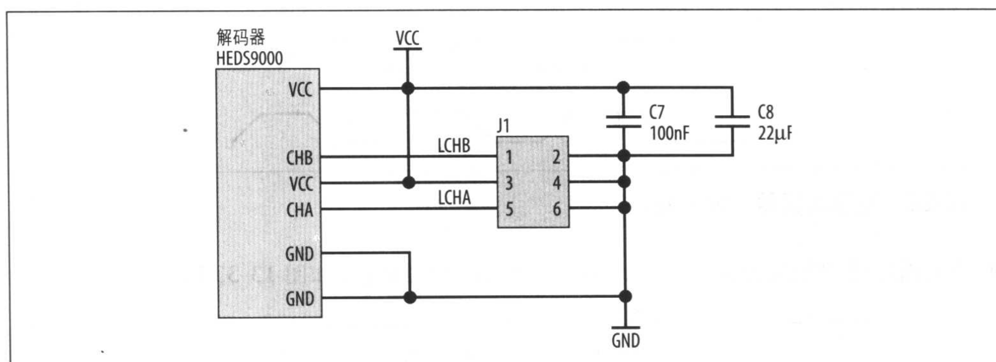


图 13-34：光学编码器电路

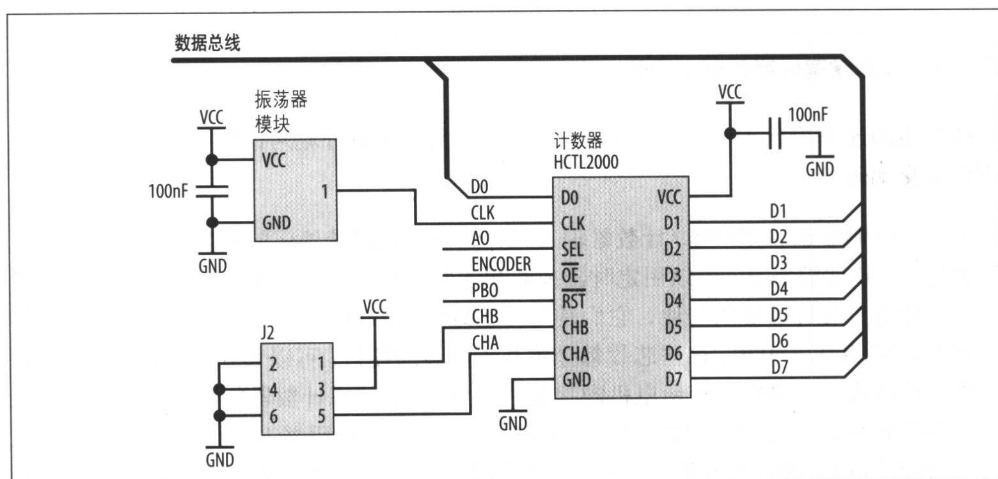


图 13-35：正交计数器电路

处理器通过 **D0 ~ D7** 这几根数据总线读取电机的当前状态。由于计数器是 12 位或 16 位，每次获得电机状态需要读取 8 位总线两次，因此，计数器在内存中有两个单元，**SEL** 输入端用来选择读取哪个字节。如果 **SEL** 为低电平，就读取高字节，反之，读取低字节。为了保证这两个字节在内存中是相邻的，用处理器的地址线 **A0** 来驱动 **SEL**。这样，两个内存单元的低字节就保存高 8 位，而高字节就保存低 8 位数据。

计数器没有这样的选择芯片，它是一个只读设备，因此其输出使能端 $\overline{\textbf{OE}}$ 就兼有片选和输出使能两种功能。$\overline{\textbf{OE}}$ 由地址译码器的输出来驱动，这个地址译码器对应着内存中计数器被映射的地址空间。当处理器对这个地址空间的单元进行读取的时候，就会触发 $\overline{\textbf{OE}}$，进而使计数器将数据送出。注意，如果处理器想要向计数器写数据，计数器就会被

选中，并对数据作出响应。所以，处理器和计数器都是将数据送到数据总线上，这有可能对两块芯片造成毁坏。现在，虽然只要编码小心点就不会有问题，不过一个大的程序中仍然可能存在疏忽，导致芯片毁坏。为了避免这种情况的出现，一个比较好的解决办法是将处理器的读选通器作为计数器的地址译码器的一部分。换句话说，只有当地址正确而且处理器也正在进行读取操作的时候，计数器才能被选中。如果处理器执行的是对计数器地址的写操作，计数器就不会被选中，对它的访问也就被忽略。

正交计数器可以让你准确地监测电机旋转的方向和速度。

控制大负载

我们前面已经知道了如何使用一个H-bridge电路芯片来对驱动电机的比较大的电压（和相应的大电流）进行转换，但是在其他的许多时候，你可能会希望让大电压接通或断开，在这一节里，我将会介绍一个实现此功能的简单方法。

Freescale MC33298是一个由微处理器通过SPI总线进行控制的芯片，它能将8个电源导通或断开。这个芯片可以控制5V~26.5V的电压，电流最大可达6A。如果你想让电路系统开启或关闭，可以使用这个芯片。最初它是应用在工业和自动化中，来控制子系统供电，比如加热器、小型空调单元、中电压照明灯、小变压器等。显然，它不能用来控制生活中使用的高压交流电，因此不要将其用来控制家用电器的电源。

图13-36所示是MC33298的基本电路原理图。

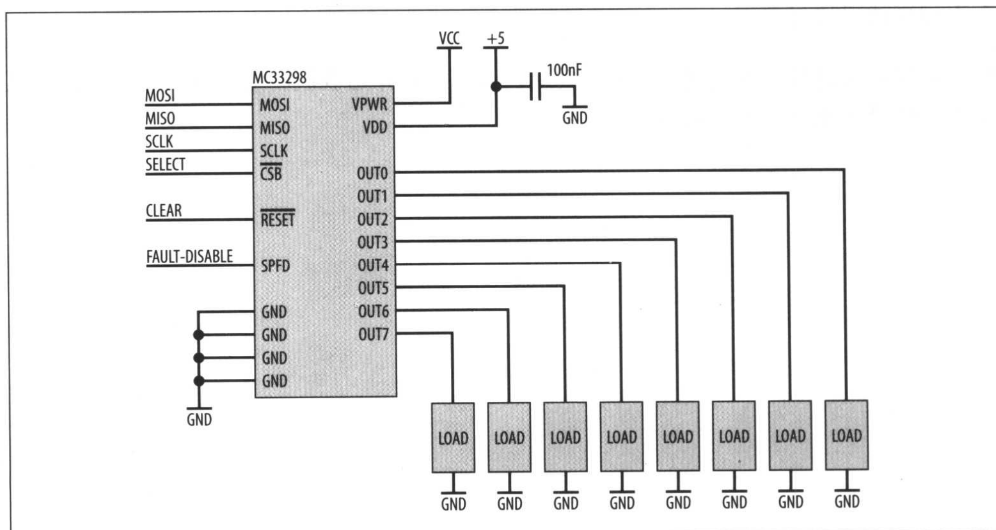


图13-36：MC33298电路

MC33298有2个电源引脚。**VDD**是一个5V的电源，用来对芯片的内部数字电路供电，它通过一个100nF的电容接地进行退耦。VPWR是外部子系统（在上图中由每个LOAD方框代表）的供电电压，它的电压值可以为5V~26.5V。这里有8个开关输出，标为**OUT0~OUT7**。当一个开关处于活动状态时，它对应的输出就连到V_{PWR}，子系统就被启动。MC33298有短路检测和关闭（可以自动重试）、过电压检测和关闭、输出电流限制、感应开关输出箝制和设备过热保护关闭功能。比较大的电流会被分到两个或多个输出上，这样就可以让多个引脚共同承受电流。当所有的输出引脚全都用上时，可以承受的最大电流可达48A，这时只受芯片总的散热量和相应的过热关闭保护功能的限制。

MC33298有一个标准的SPI端口，这可以让它连接到大多数微处理器，进而受到控制。芯片的SPI信号**MOSI**、**MISO**和**SCLK**直接连到处理器的SPI引脚上，它的选择输入端$\overline{\textbf{CSB}}$（由处理器的一个数字输出控制）用来在一个SPI传输中选择芯片。通过对芯片的$\overline{\textbf{RESET}}$输入端输入有效信号可以将芯片复位，并关掉所有的输出。同样，这个复位输入也可以由处理器的数字输出来控制，这样就可以在软件控制下将芯片关闭。MC33298支持SPI的菊花链连接，因而可以将多个芯片连接到一起使用。

SPFD是短路故障保护禁止引脚，向它输入高电平可以让内部的过电流检测电路无效。当对一些负载（比如照明灯）进行控制的时候，会有很大的瞬时电流，这通常会被MC33298判断为一个故障，并关闭输出。**SPFD**引脚可以让过电流保护功能失效，进而对这类负载进行控制。即使过电流保护功能被禁止，MC33298仍可以受到保护。如果大电流持续时间太长，它的过热关闭电路将会起作用，以防止芯片被毁坏。**SPFD**可以由处理器的数字输出驱动，但应该谨慎使用。在正常的工作情况下（过电流保护开启），这个引脚应该保持低电平。

讨论完嵌入式计算机的各种I/O选项之后，从下一章开始我们将看到几种处理器，并且说明如何设计整套嵌入式系统。

第14章

PIC 微控制器

ENIAC 机上的一个计算器含有 18000 个真空管，重达 30 吨；
但将来的计算机也许只有 1000 个真空管，而重量不过 1.5 吨。
——《通用机械学》，1949 年 3 月刊

本章将介绍 Microchip PIC。为了便于对微处理器硬件进行研究，我们先通过设计一款小型计算机来学习制造计算机硬件的基础知识，这款小型计算机基于一个 8 引脚的 PIC 处理器。同样的设计原则也可以应用到 AVR 和很多其他的微控制器上。正如你将要看到的那样，因为PIC处理器的结构很简单，所以用它来构建计算机也是很容易的。稍后，还将介绍一种中等规模的PIC处理器，并介绍用它来设计嵌入式系统所需要的知识。在进行设计之前，首先快速浏览一下 PIC 的体系结构。

两款处理器的发展史

在20世纪70年代末，通用仪器（General Instruments）公司研制出一种16位的处理器，即 CP1600。但由于 Intel 8086 和 Motorola 68000 芯片的竞争，CP1600 很早就被淘汰掉了，因而没有给人们留下什么较深的印象。CP1600的缺陷在于它限制了I/O的能力，因而通用仪器公司又设计了一个小型的扩展处理器，来用作 I/O 控制器。该I/O 控制器不但能为CP1600提供I/O，而且作为一个处理器，它还能提供某种程度的智能控制，这款处理器称作外围接口控制器（Peripheral Interface Controller，PIC）。CP1600芯片慢慢地消亡，最终被人们遗忘了，但是作为它的一个小部件的PIC却逐渐成长起来。在20世纪80年代中期，通用仪器公司的微电子分部被合并到Microchip公司，其核心产品就是 PIC 处理器。PIC 有着很广泛的用途，在索尼公司的 PlayStation 控制器、儿童玩具、消费器具以及工业系统中都能看到它们的踪影。

最初的PIC体系结构中只有一个累加器（即工作寄存器），它是一个25~368字节大小的RAM。程序计数器的最低有效位、状态寄存器和各种控制器都被映射到RAM的低地址空间，可以由标准内存移动（move）指令所访问。RAM的高位地址空间用来存储数据。Microchip公司把RAM空间称作“寄存器”，其实它们仅具有真正寄存器的部分功能，其主要用途是数据存储。

处理器芯片有一个固定大小的堆栈，根据处理器的不同，堆栈的输入项数目为2~8个。该堆栈的唯一作用是存放子程序调用或中断的返回地址。系统还使用了一个独立的文件选择寄存器（File Select Register，FSR）作为映射到RAM地址空间的索引寄存器，通过使用FSR可以实现一定的变址寻址，FSR也可以作为用户数据的伪堆栈。

除了少数的特例，PIC控制器没有外部总线，是位于一片独立的芯片上的完备的计算机。通过使用处理器的外设接口（SPI和I^2C）或数字I/O端口，可以对其功能进行扩展，但这种扩展很有限。对于那些对体积和功耗要求很高的应用，PIC具有绝对的优势。在设计中能够采用微小的计算机系统是很有利的，而且对于采用电池供电的应用也是理想的，因为小巧的计算机系统能够（几乎）用尽电池中的每一个游离电子。

PIC还非常健壮。要毁坏一个PIC芯片也不是那么容易的。我有一个客户曾在无意中接反了基于PIC的计算机的电源和接地，直到一周后他才发现。然而，令人惊讶的是，它却仍可以正常工作（只要合适地供电）。还有一次，为了测试基于PIC芯片的数字记录仪的性能，我们把它放到印度太平洋快递公司（Indian Pacific Express）的快递车上。这辆快递车是行驶在悉尼和佩思之间的长途客运列车，途经澳大利亚中部的大沙漠。可是在试验的过程中不幸的事情发生了，快递车卷入到一起严重的车祸当中。当时的情况是这样的，由于信号灯的操作失误，使一辆班车追尾撞上了快递车，完全撞坏了最后几截车厢。而数字记录仪恰好是被安装在列车的尾部，它完全承受了碰撞所产生的冲击力，可是当一切恢复以后，数字记录仪竟然仍能正常工作。PIC芯片这个小小的处理器的生命力之强由此可见一斑。

PIC处理器的很多方面类似于RISC结构，它所采用的体系结构是哈佛结构，因而具有独立的数据空间和代码空间。数据空间的字长是8位，而代码空间的字长介于12~16位之间（具体由PIC的类型决定）。数据地址空间被映射到多个存储体中，包括大部分的控制寄存器。可是由于PIC控制器只有一个累加器、存储体以及有限的寻址模式，这使得有一定比例的给定程序消耗在来回转移数据上，但大多数其他的处理器并不如此。在小规模的简单应用方面，PIC处理器很有优势。然而，超低功耗的诱惑使人们试图用它来解决一些极其复杂的算法问题。不过，为PIC处理器写一个复杂的软件有时就好像解决只有一根栓的汉诺塔问题一样不可能。它的确是个巨大的挑战！很多PIC程序员都希望有更多的内存空间、更多的累加器，Microchip公司的新型*dsPIC*体系结构比标准PIC

有了显著的改进，它的诞生使得全世界的PIC开发者都兴奋不已，它是一个先进得多的处理器。

Microchip 软件开发环境（MPLAB）提供了一个汇编器（assembler）、一个仿真器（simulator），以及一套用以把代码烧录到处理器中的软件。MPLAB 可以从 Microchip 公司的网站上免费下载。PIC 支持部分的商用 C 编译器，但没有为 GNU C 编译器提供接口（但在写本书的时候，有传言说在新一代的 *dsPIC* 结构中将为 GNU C 编译器提供接口）。

对于许多简单的数字应用，一个小的微处理器比一个离散的逻辑电路更适合，因为它能执行软件。正因为如此，在执行某种任务时，它才能降低硬件的复杂度。下面就让我们体会一下用 PIC 来构建一个小型的嵌入式计算机是多么的简单。

一个简单的实例

PIC12C508处理器是一款小型的8引脚计算机（内部只有512字的程序存储器和25字节的内部RAM），用于实现最简单的控制功能。它完全可以用于任意简单的应用，如对数字输入的监控和开关功能。PIC12C508 处理器的 I/O 引脚可以用来连接一个 SPI 或 I^2C 接口，或是使用 PWM 来控制电机。

图 14-1 所示是基于PIC12C508处理器的小型计算机的电路原理图。PIC的数字I/O信号通过引脚7的连接器输出。在设计的实现中，如果在可能的地方都使用表面贴片元件的话，那么连接器将是 PCB 上最大的元件！

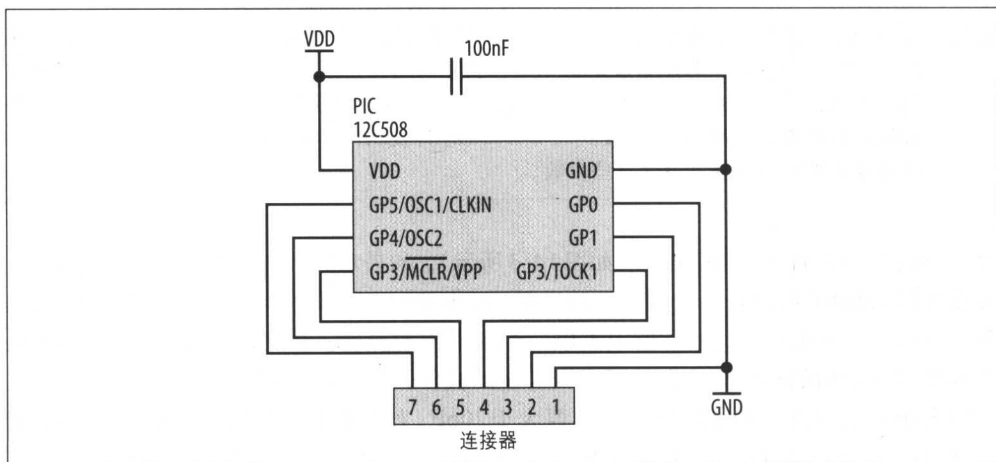


图 14-1：小型的 PIC12C508 计算机

这种独特的PIC处理器还内置了一个频率为4MHz的RC振荡器，因而在使用这种处理器时不再需要任何外部振荡电路。图14-2所示是同一种基于PIC的设计电路，但在这个电路中使用了一个32kHz的外部晶体作为它的振荡器，使用其振荡器可以大大减少处理器的功耗。对那些使用电池供电的系统而言，这点尤为重要。

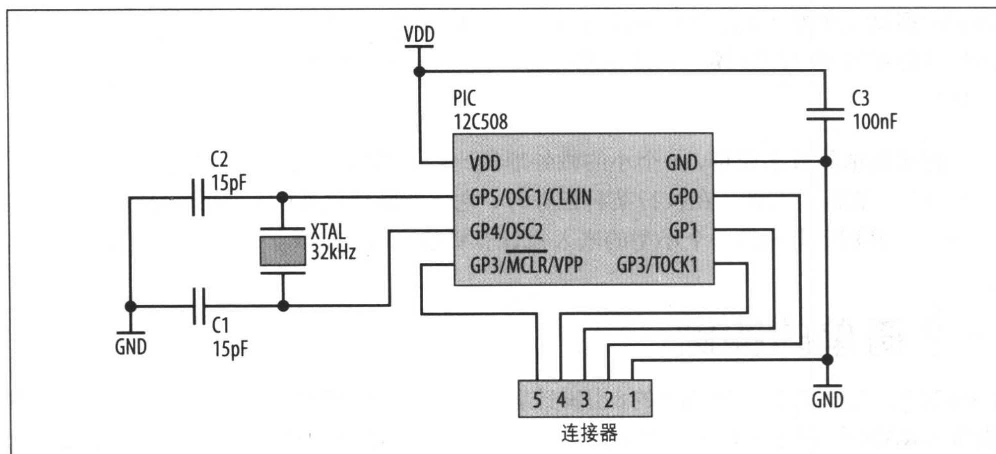


图 14-2：一种基本的 PIC12C508 计算机（增加了电源）

由于晶体振荡器会产生多余的高次谐波，所以系统用两个15pF的电容去除这种影响。电容的大小取决于所用晶体的频率和型号，处理器技术手册上列出了不同频率的晶体所需的电容值。

注意： 在使用不同的振荡器资源之前，需要对PIC处理器进行合理地配置。如果PIC处理器使用的是32kHz的晶体，芯片必须置成“LP模式”。如果PIC处理器使用更快的晶体（频率超过455kHz），芯片的配置模式为“XT模式”。内部RC振荡器选择“INTRC模式”，而外部振荡器则要求使用“EXTR模式”。Microchip开发软件（MPLAB）可以使你在往处理器中烧录软件时很容易地设置这些参数。

另一种时钟源来自外部RC电路（如图14-3所示）。如果对时序精度的要求不是非常高，那么使用外部RC电路是一种很经济的选择。振荡器的实际频率由以下几个因素决定：电阻、电容、电源电压、元器件的误差以及当前操作的温度。显然，这表明RC振荡器只能提供一个近似的操作频率。为了得到稳定的操作，Microchip公司有以下建议：电阻值的大小介于3kΩ~100kΩ之间，电容要大于20pF。如果你打算使用外部RC振荡器，请参考Microchip公司的技术手册。Microchip公司已详细列出RC元件的选择信息，以及要考虑到的电压和温度的影响。

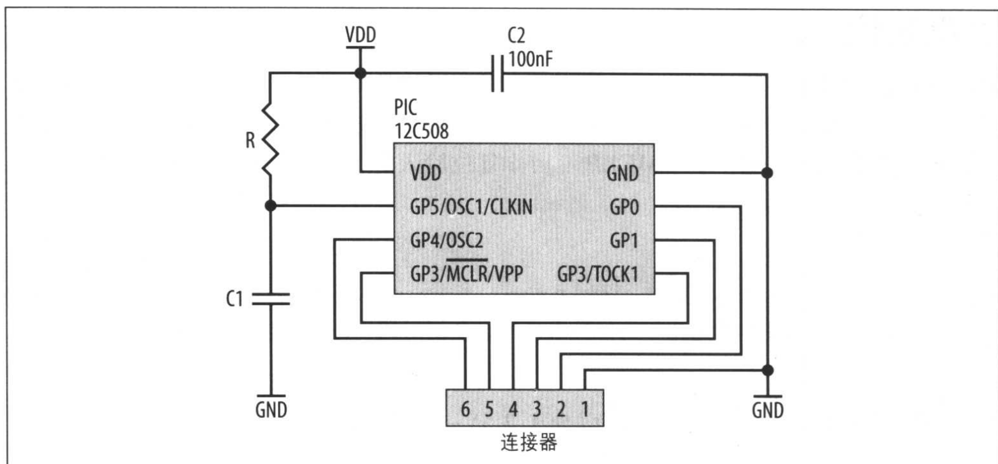


图 14-3：外部 RC 振荡器

变频振荡器

使用外部RC振荡器的一个明智的方法是把你的计算机配置成变频的。可以通过在振荡器输入端与I/O引脚之间增加一个上升电阻R1来实现，如图14-4所示。正常情况下，I/O引脚是作为输入端。但如果把I/O引脚作为输出端并置为高电平，则电阻R1可以有效地与电阻R并联。由公式$R_{TOTAL}=1/(1/R+1/R_1)$可知总阻抗增加了，而振荡器频率则相应地减少了。这是一种很有用的方法，它可以使系统在软件的控制下减少功耗。

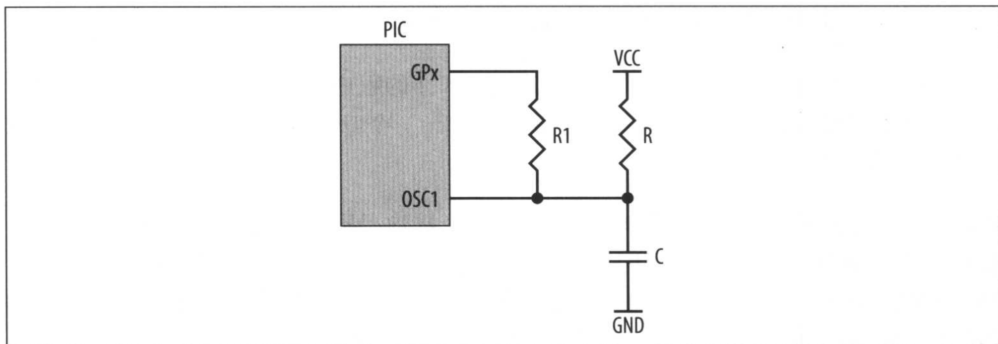


图 14-4：变频 RC 振荡器

当使用外部RC电路驱动内部振荡器时，可以使用一条专用的PIC I/O线路（**GP4**）。

上电复位

这款PIC处理器不需要外部复位，在设计中所使用的是处理器的内部上电复位电路（译注1）。而且，在复位输入端（$\overline{\mathbf{MCLR}}$）甚至不需要外置电阻，而是由处理器使用一个弱上升电阻来达到这一目的。当$\overline{\mathbf{MCLR}}$不用作复位输入时，可以作为一个通用的输入端来使用。

警告：对于其他的一些PIC处理器，$\overline{\mathbf{MCLR}}$需要一个上拉电阻或是直接连到V_{DD}。否则，它既不能工作，也不能作为通用目的的输入。所以在使用PIC处理器前请查看相关技术手册。

PIC12C508的电源电压（V_{DD}）可以介于2.5V~5.5V之间。

以上讨论了PIC12C508计算机系统的基础知识，这与我们下一章要研究的AVR计算机并没有太大的区别。真正的不同在于以下几个方面：内部体系结构（指令集）、操作电压上的细微差别以及接口功能。你将看到，把这些小器件放入你的嵌入式系统中并不需要花费太大的力气。

一个更大的PIC处理器

在这一节里，我们要介绍PIC16C73处理器。这个中等规模的处理器与我们已经看过的简单的PIC处理器并没有很大不同。唯一的不同点在于它有更多的引脚、更多的I/O以及更多的功能。设计PIC17和PIC18处理器与构建基于PIC16系列的机器并没有什么太大的不同。在此你所学到的内容可以应用到许多其他PIC系列处理器上。

这种处理器的电路原理图如图14-5所示。它有4K字的程序地址空间、192字节的RAM、多种I/O子系统（如3个定时器模块）、SPI、I^2C、一个UART、5个模拟输入通道以及多达22个的数字I/O引脚。

PIC16C73处理器有一个电源引脚（**VDD**）、两个接地引脚（**VSS**）。像通常一样，电源使用一个小电容（C3）接地用以退耦。此外，还需要的就是一个时钟发生器，在本例中使用的是晶体X1，以及两个退耦电容C1和C2。时钟可以很方便地采用我们在12C508 PIC中用到的RC电路。复位输入端（$\overline{\mathbf{MCLR}}$）直接与电源相连，以使其永远处于无效状态。在本例中，我们使用的是处理器的内部上电复位电路，所以不再需要外部复位。通常都是使用一个上升电阻来连接无用的输入端，使其处于无效状态（如$\overline{\mathbf{MCLR}}$）。不过，

译注1：当V_{DD}引脚检测到正跳变时，发生上电复位。

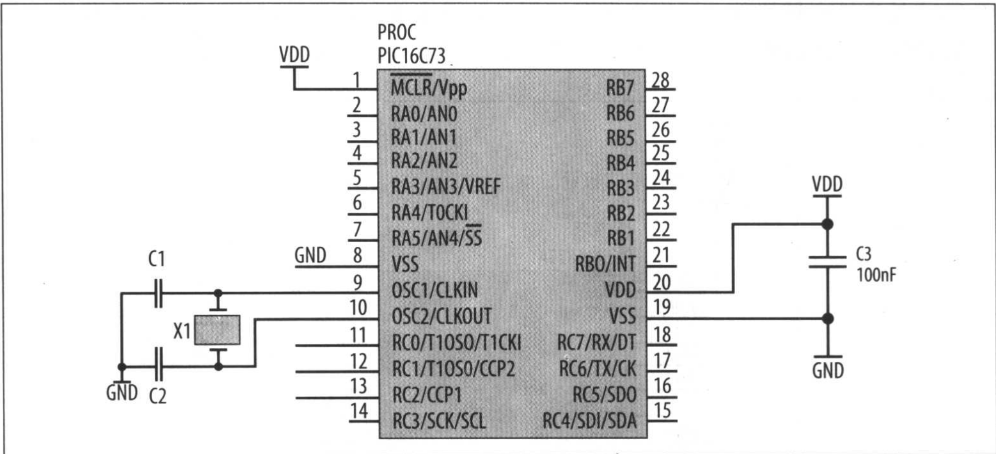


图 14-5：PIC16C73 处理器及支持元件

在本例中一个上升电阻会影响内部上电复位的跳变，使处理器难以正确启动。因而，在本例中最好将其略去。

端口 A（**RA0~RA5**）可以作为一个模拟输入端口或一个通用数字端口。端口 B（**RB0~RB7**）是一个具有微弱内部上拉电阻的通用数字端口。端口 C（**RC0~RC7**）可作为一个数字端口，或是提供定时器、PWM、一个串口、一个 SPI 或 I^2C 接口。你可以根据需要在自己的设计中使用某些或所有引脚，连接其他子系统。

以上的基本设计连同相应的技手册，可以很容易地应用到你所遇到的绝大部分 PIC 处理器中去。

基于 PIC 的环境数据记录器

现在来看一个基于 PIC 处理器的完整系统。这一节所要介绍的是我的 DL4 数据记录器的简化版。这个数据记录器被设计成使用最小的电量记录数据（至少一年）。它具有 1MB 的非易失性存储器，可以在没有电量的状况下保存数据 20 年之久。尽管此处所安装的是光线和温度的传感器，不过很容易就可以将此设计用在你想要使用的任何传感器上。此数据记录器相当小，整个设备安装在电路板上之后比你的小指还要小。

该设备所采用的处理器是 PIC16LF873A，这是 PIC16C73 的另外一个版本。L 代表低功耗，F 代表这是一个基于闪存（而不是 EPROM 或 OTP）的元器件，可以在电路上直接进行重新编程，这让调试的过程变得简单许多。8 代表处理器内部含有 EEPROM 作为参

数的非易失性存储器件，可用来保存使用者的设定值和机器的状态。最后，A 代表这是个硅片的第二次改版。该处理器的基本电路及其支持元件如图 14-6 所示。

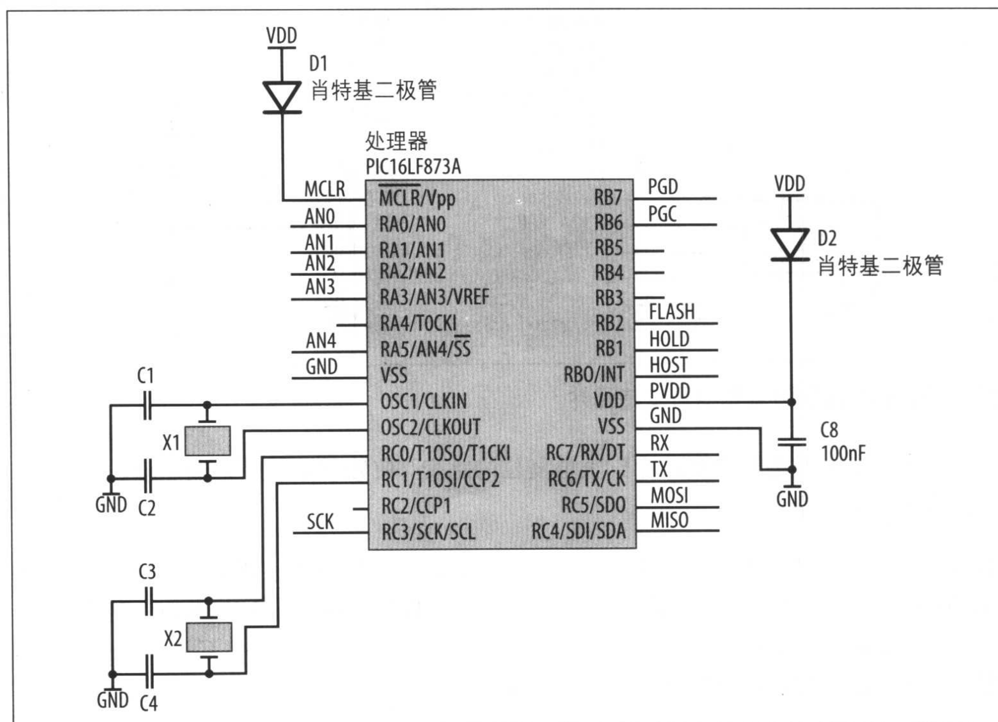


图 14-6：数据记录器处理器及支持元件

注意，此处理器的电源引脚被连接到线路标记 **PVDD**，而不是系统的 **VDD**。因为此处理器可以在电路上直接进行重新编程（烧录），所以必须将此纳入我们的设计中。烧录期间，烧录器会将自己的电源（+5V）提供给处理器。数据记录器中所用到的闪存芯片，其额定电压为 3.3V，烧录器所提供的电压是 5V，这就可能对闪存芯片造成损害。因此，我们使用了一个肖特基二极管（D2），以便在烧录期间将系统电源与处理器隔开。非烧录期间，D2 会导通并提供电源给处理器。烧录期间，D2 不会导通，数据记录器的其余部分将会停在未供电状态。

同理，我们使用了另外一个肖特基二极管（D1）来隔开复位引脚和系统电源。非烧录期间，D1 会导通，使得 $\overline{\mathbf{MCLR}}$ 被提升到 **VDD**。然而，烧录期间 D1 会隔开 $\overline{\mathbf{MCLR}}$ 和系统电源，这让烧录器得以将此复位引脚提升到所需的较高电压。

此处理器具有两个晶体：X1 和 X2。X1 用来给处理器提供时钟，让它得以进行内部的操

作。X1 的范围为 32kHz~20MHz，依应用而定。X1 会影响到数据记录器的功耗。晶体越快，机器所耗用的电量就越多。欲进行超低功率的运行，32kHz 的晶体是最佳选择。然而，这么做也有一个缺点。因为用来驱动 UART 收发器的内部振荡器，在晶体很慢的状况下，将会限制我们所能达到的波特率。使用 32kHz 的晶体时，波特率最大只能达到 300。在这样的传输率下，从数据记录器下载 1MB 的数据，将需要花费 7 小时 42 分钟的时间！因此，你应该根据自己的需要为 X1 选择最恰当的值。如果你可以容忍 7 个小时的下载时间，并且想让你的电池获得最久的使用寿命，就使用 32kHz 的晶体吧。如果电池的寿命不是个问题，则可以使用速度较快的晶体。

X2 用来驱动处理器内部的 TIMER1 子系统，这个子系统可用于计时功能以及采样的过程。尽管我们可以使用处理器的主振荡器来驱动内部定时器，但是此振荡器会在 `SLEEP` 指令执行期间遭到关闭（内部的看门狗电路可用来唤起处理器）。因此，最好在进行定时功能时为 TIMER1 使用第二个晶体，这样即使在休眠期间 TIMER1 仍会继续运行。

稳压电路如图 14-7 所示。这是一个标准的 MAX604 电路，可以为 VDD 提供 3.3V 的电源。因为我们使用电池作为电源，所以输入端可以省略掉电容。电池的选择很多，我喜欢用 Energizer 的 EL123 电池。该电池的体积相当小（长度是 AA 电池的三分之二，但电量略多一点），却可以让数据记录器运行超过一年的时间。

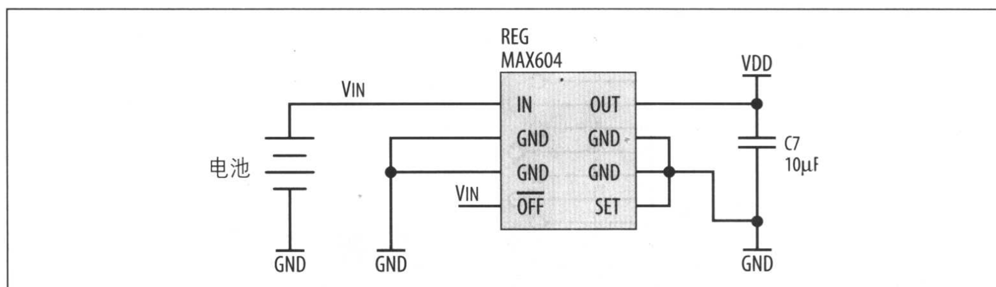


图 14-7：数据记录器的电源

这个稳压器对数据记录器而言很重要，原因有二。首先，它可以确保数据记录器的电源电压是恒定的。这一点很重要，因为此电源电压将会作为模数转换的参考电压。尽管直接以电池供电，处理器的运行绝对没有问题，但是电池的电压在使用的时候会逐渐下降，这样从传感器读回的值将会渐渐变得没有什么意义。换言之，尽管数据记录得相当成功，但是却毫无用处。

图 14-8 所示为 ST-Electronics 所制造的非易失性闪存芯片，可用来保存数据。该闪存使用简单的 SPI 接口与主处理器连接。闪存的片选输入（**FLASH**）通过处理器的 I/O 引脚（**RB2**）来控制。存取期间，闪存的 **HOLD** 输入具有“暂停”的功能。这会使得处理器

暂停存取闪存的动作，以便对其他外设进行SPI传输。数据记录器目前尚未使用此功能，将它纳入设计之中可作为未来扩充性能之用，例如用在基于SPI的数字传感器上。最后，片选输入（**FLASH**）会被提升到高电平，以避免闪存在数据记录器开机的时候被无意识地选中。

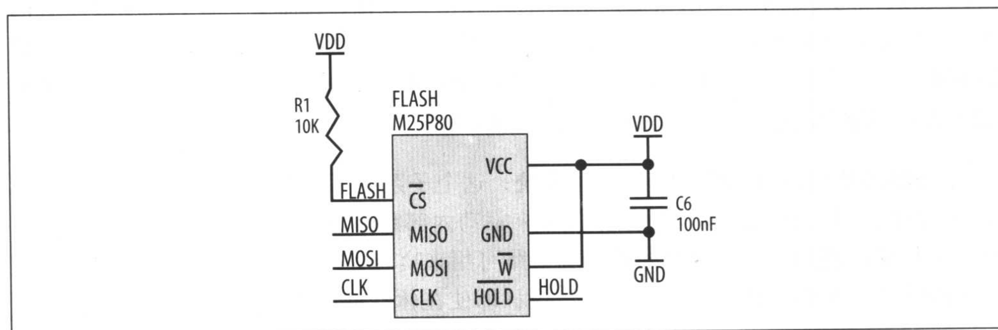


图 14-8：数据记录器的非易失性闪存

图 14-9 所示为数据记录器用来连接外界的接口。DL4 数据记录器使用了一个小型的 Harwin 连接器，不过你可以根据自己的需要使用任何连接器，即使是 IDC 连接器。

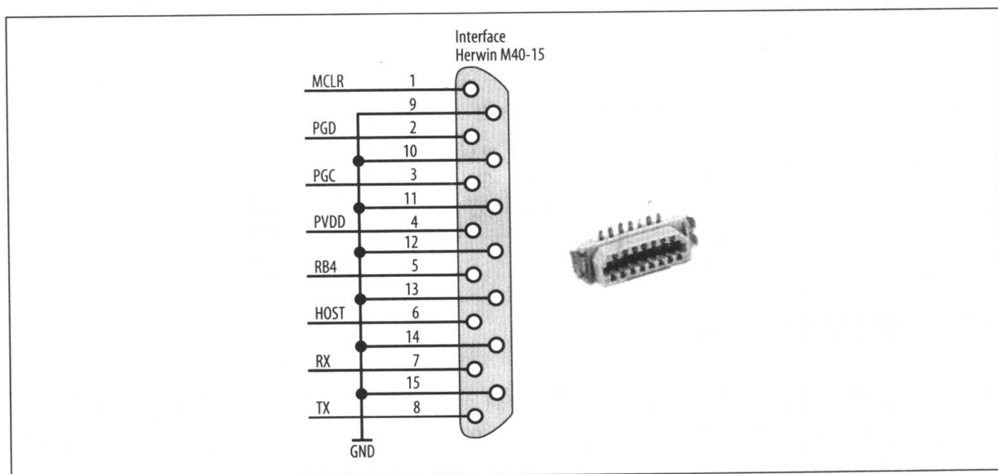


图 14-9：数据记录器的连接器

这个连接器可用来配接两个独立的设备。第一个设备如图 14-10 所示。这是一个自制的转接板，可让数据记录器插入 Microchip 的 PICSTART Plus 烧录器，以便烧录新的程序代码。该连接器只是将烧录期间所需要的信号（**MCLR**、**PGD**、**PGC**、电源与接地）对

应到基于DIP封装的PIC的等效引脚上。基本上，它是将数据记录器的连接器转换成基于 DIP 封装的 PIC 引脚。这个转接板之下具有可以插入烧录器的引脚。

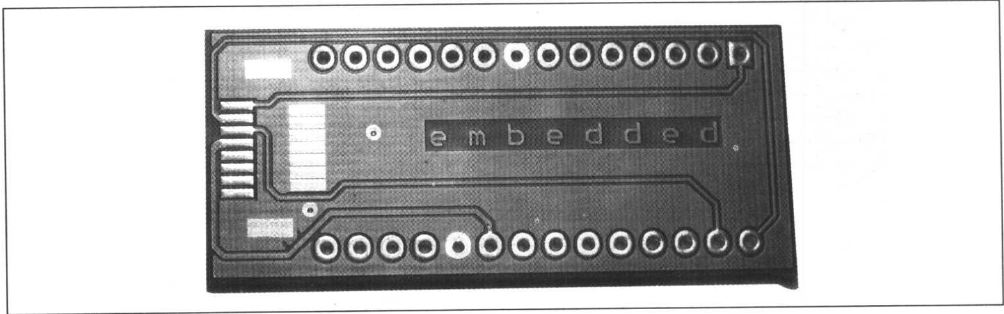


图 14-10：DL4 数据记录器的烧录器转接板

数据记录器可插入的第二个设备就是 RS-232C 转接模块，如图 14-11 所示。

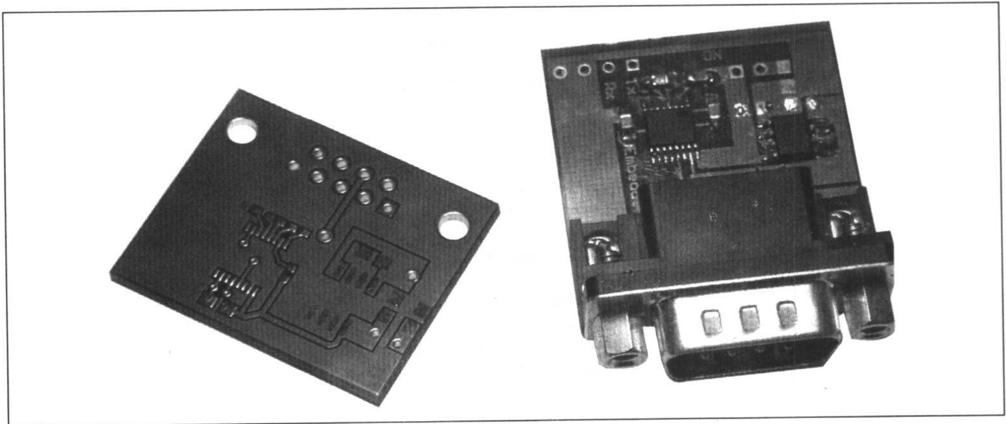


图 14-11：串行转接板

该模块让数据记录器得以在需要动态设定和取回数据时，被连接到上位计算机。此转接板具有一个 Maxim RS-232C 电平转换器、一个稳压器（它们就放在电路板的顶层）以及一个 DB-9 连接器。该模块的电源取自上位计算机串行口的 **RTS** 信号，因此不需要提供外部的电源，其电路原理图如图 14-12 所示。注意，这是一个独立于数据记录器之外的电路。

之所以要使用二极管 D2，是因为 **RTS** 具有负电压和正电压。D2 可避免 **RTS** 变成负电压时损坏稳压器。在正常操作期间，上位机在软件的控制下会把 **RTS** 置为 +12V，这就

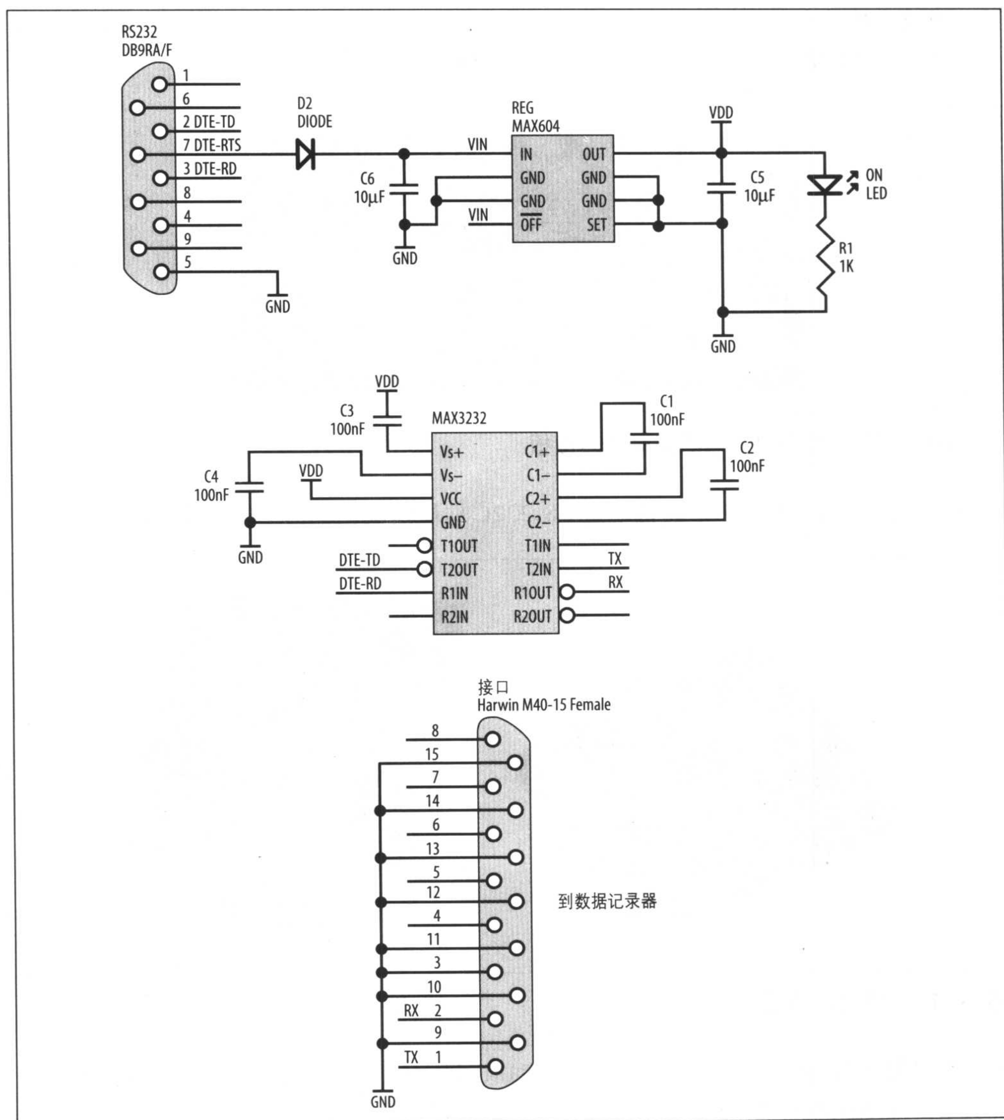


图 14-12：串口转接板电路

是串口转接器的电源。稳压器会把该电源转换成 +3.3V，并提供给 MAX3232 和 LED。注意，数据记录板上的 Harwin 连接器的 6 号引脚会被连接到标记为 **HOST** 的线路。此引脚会被对应到转接板上 Harwin 连接器的引脚 3，而且会被接地。当数据记录器插入转接模块时，**HOST** 会被置为低电平，但是其他时候它是高电平，因为 PIC 内部有上拉电阻。因此，数据记录器上所执行的软件便可轻易判断出目前上位计算机是否存在。这样

数据记录器的UART就可以在适当的时候进入省电模式。此外，对软件而言，它的行为如同一个简单的开关，可据此在“与上位机对话”模式和“数据记录”模式之间切换。值得注意的是，在电路图上，Harwin连接器因为引脚数的关系，所以看起来比DB-9连接器还要大。然而，与DB-9放在一起时，Harwin连接器其实是很小的。

最后，图14-13所示为数据记录器的传感器。这两个传感器的相关细节参见第13章。当然，你可以使用其他的传感器，或是提供一个连接器接口可替换的传感器模块或外部的模拟子系统。

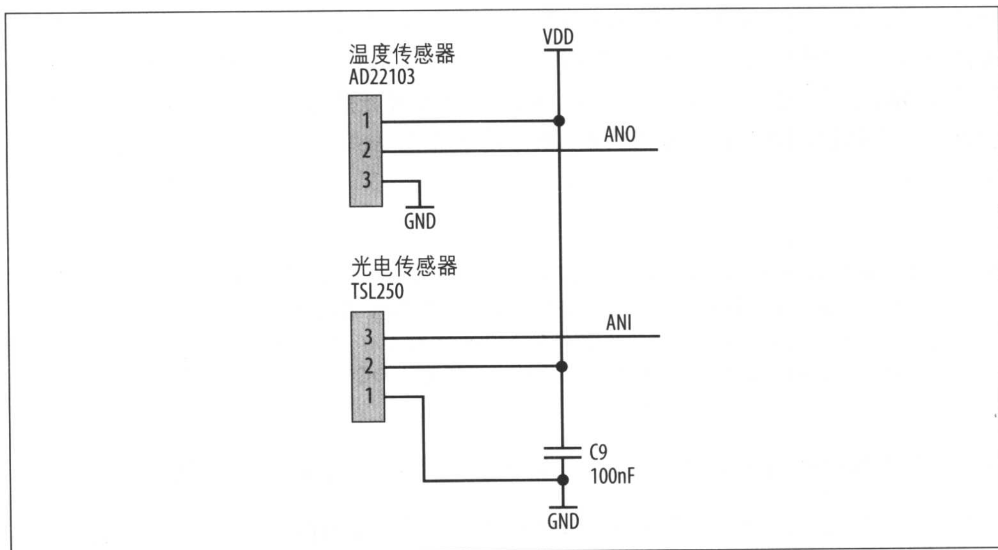


图14-13：数据记录器的传感器

用PIC来控制电机

现在来看一下PIC处理器另外一个完全不同的应用：经由使用者的输入来控制电机。尽管这一节所要介绍的设计针对的是特殊的应用，不过你可以轻易地将它应用在其他需要控制小型直流电机的工作上。

我的小外甥喜欢玩模型火车，铁路上的标准控制器是一个相当简单的设备。火车头的速度是由轨道上的电压来控制。电压越大，火车的速度就越快；电压越小，火车的速度就越慢直至停下来。这无法实现精细控制。我决定通过一些高科技手段来解决此问题，并且使用PIC来设计一个基于微处理器的控制器。我将会跟大家分享此设计。

注意：这些年来，我设计过不下数十种嵌入式系统，从极小的控制器到小型的并行超级计算机都有。但是我必须要说，当我在对火车控制器进行调试时，得到的是前所未有的乐趣。

此设计利用到了第13章所介绍的PWM技术来控制火车头的电机。因为PWM使用振幅固定的脉冲来驱动电机，所以低电压时火车头停下来的问题不见了。此外，因为PWM受软件的控制，所以可以达到非常低的速度。PIC处理器内部集成了两个PWM发生模块。使用PIC来产生PWM时，其实只是编写一些简单程序的工作而已。

硬件的设计需要进一步考虑到动量控制和刹车等基本概念。动量控制让你得以在受控火车的质量固定的情况下（译注2），通过打开节流阀来让火车渐渐达到它的速度，当你关闭节流阀时，火车就慢慢停下来。刹车可以加快火车停止的过程。这听起来好像很复杂，但是当你以微控制器来做时却很简单。实际的工作都是由软件来完成的，你愿意怎么做就怎么做。

注意：模型火车的控制还有--个更复杂的方法，称作数字指令控制（Digital Command Control，DDC），该技术把轨道当成局域网来用，而且会传送控制包给火车头。这是一个已公开的标准，相当复杂。相关的细节可以在 *http://www.dcc.info* 上找到。此处的设计并不像DCC那样复杂，也不会与DCC相兼容。不过，它绝对比DCC便宜，而且容易实现。

处理器的电路原理图如图14-14所示。它与用于数据记录器中的并没有什么不同。

你会注意到该系统中只有一个晶体，因为此处并不需要定时的功能。有两个状态LED，通过软件来为用户提供反馈。端口B上还有几个引脚置空（**RB2**、**RB5~RB7**），有需要时，可以自行添加额外的LED。**BRAKE**和**FLAG**是处理器的输入端，前者来自一个按钮开关，而后者来自H-bridge芯片的输出，用以指示是否发生过电流错误。

你还会注意到，此处并没有像数据记录器那样提供在电路上直接烧录的功能。针对这个特定项目，我特别使用DIP封装的PIC处理器，需要烧录PIC时，只需要把它从电路板上的插座上取下，然后插入烧录器即可。如果你想要在电路上直接烧录芯片，只需要使用跟数据记录器一样的线路就可以了。

晶体的频率由你自己决定，不过你所做的选择将会导致一个有趣的结果。晶体的频率与处理器所能产生的PWM频率有直接的关系。来自晶体电路的时钟信号在进入处理器的

译注2：牛顿认为动量p应该与质量m成正比，也应该与速度v成正比，于是可定义成p=mv，单位是kgm/s。

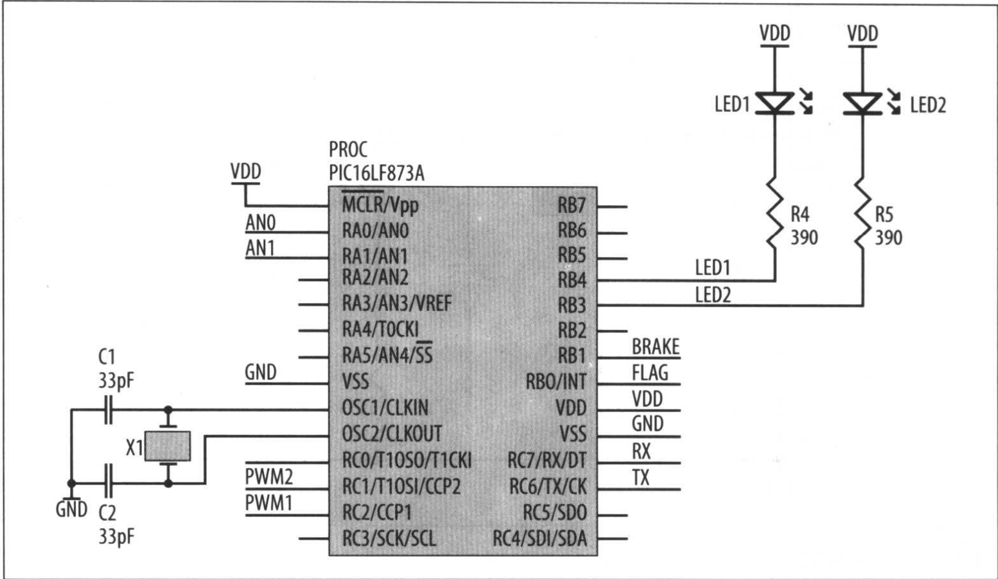


图 14-14：处理器

PWM 模块之前被除以 4。PWM 模块中的寄存器会将大小再往后调，以便产生更慢的 PWM 频率。商用的火车控制器使用的是 16125Hz 的 PWM 频率。只要频率比该值低，你将会听到电机的噪音；只要频率高于此值，则会损耗扭力。所以，要达到 16kHz 的 PWM 频率，将需要一个频率高于 64kHz 的晶体，然后通过 PWM 模块的寄存器将大小再往后调到适当的值。

注意：现在假设你不想让火车头的声音听起来像电机。如果你的铁路模型使用的是内燃机，可能会想要使用频率为 32kHz 的晶体（这是一个很容易取得的手表晶体），然后往后调整它的大小，让 PWM 频率达到数十 Hz 的程度。这样的频率会使电机产生隆隆声和震动的噪音，与真正的内燃机像极了。

稳压电路如图 14-15 所示。这个稳压器我选用了标准（并且很便宜的）的 LM7805，它可以接受 +7V~+35V 的输入电压，提供恒定的 +5V 输出电压。因为模型火车头电机的最大额定电压为 +12V，这是实际提供给该系统的电压。稳压电路图中还包含了一个 LED，用来指示电源已经开启，额外的退耦电容用来消除系统中的数字干扰。

图 14-16 所示是用来驱动火车的各种控制。节流阀和动量控制只是 50kΩ 的电位计，它们的作用就是分压。这些电位计的滑片可以提供范围为 0V~+5V 的电压给 PIC 的模拟输

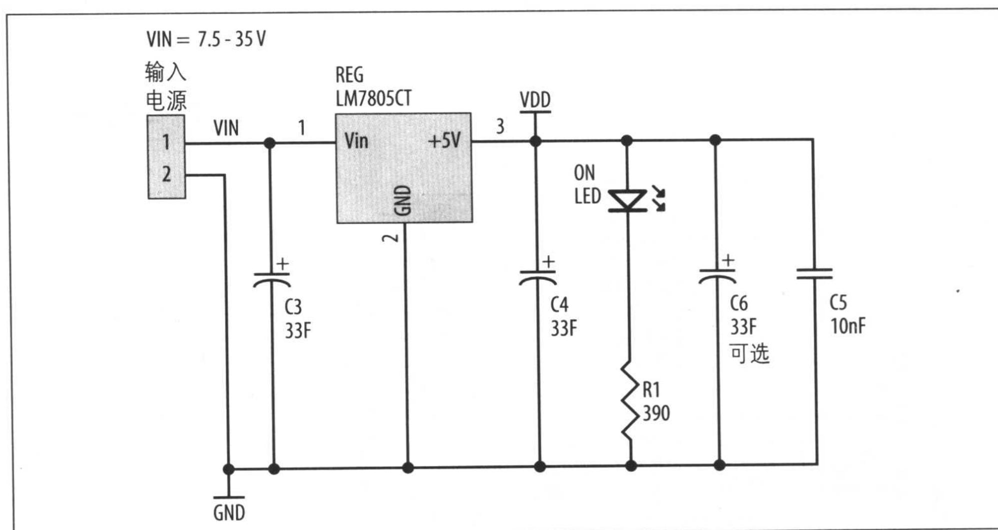


图 14-15：稳压器

入端。因此，软件可以轻易取得控制器的位置。方向和刹车控制只是按钮。方向控制会被连接到H-bridge芯片（之前讨论过）的输入端，刹车控制会被连接到处理器的数字输入端。方向控制具有100kΩ的上拉电阻，而刹车控制则会使用处理器端口B的上拉电阻。

图 14-17 所示为H-bridge芯片，该芯片会把处理器的PWM输出的电压提升到足以驱动模型火车头上的直流电机，或是任何小型的直流电机。

H-bridge使用**VIN**（+12V）作为系统的电源，因为它必须提供该电压给电机。正如之前所说的，**DIR**（方向）输入端只是按钮，用来决定输出的极性。或者这个方向开关可以连接到处理器上的空闲数字输入端，然后使用数字输出端来驱动**DIR**。因为软件只是将输入对应到输出，所以还是跳过PIC直接连接会比较简单。

PIC的PWM1模块输出会被直接连到H-bridge芯片的PWM输入端。H-bridge芯片会将5V的振幅（来自PIC的PWM信号）转换成12V的振幅（PWM的输出其极性由**DIR**来决定）。

注意： 你可能注意到了，PIC具有两个PWM模块。如果你希望以第二个模块来控制第二个电机（火车或其他），只需要复制H-bridge电路，并将它连接至PWM2即可（别忘了在电路原理图编辑器中使用不同的线路标识，以便正确分开电路）。

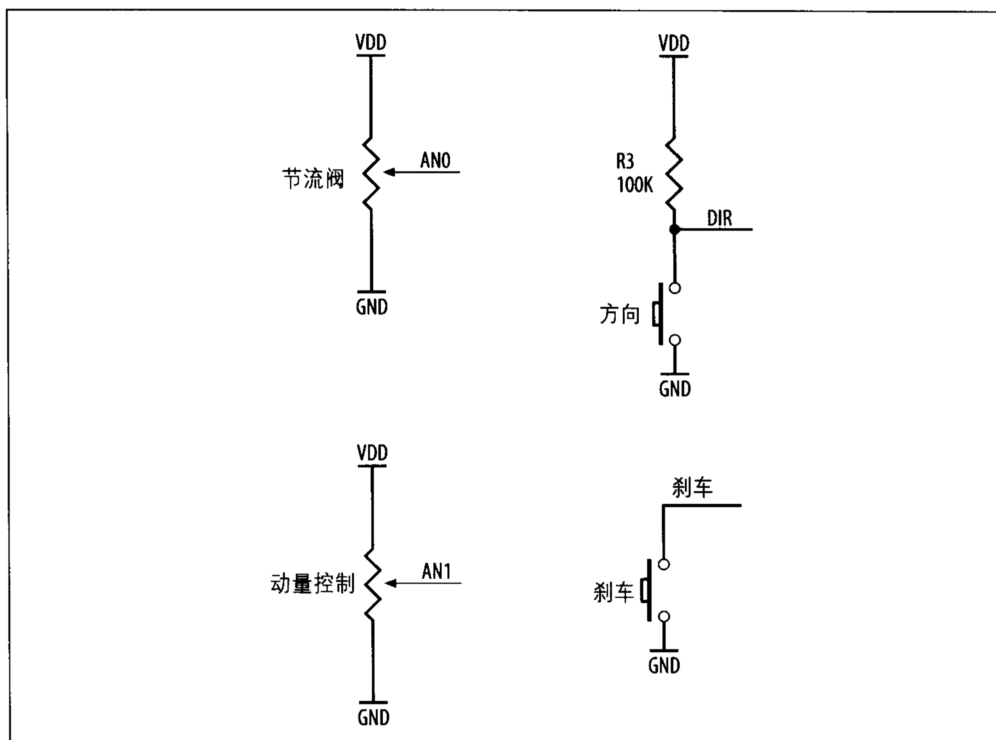


图 14-16：各种控制

H-bridge芯片集成了内部电流侦测，所以温度太高时它会将自己关闭（如果输出被短路在一起，就会这样）。发生此类问题时，**FAULT**输出端便会指示此状况。这个低电平作用的输出可用来驱动一个LED，也可以连到处理器的一个输入端，让软件得以获知此问题。

最后，我们可以将一个串口加入控制器，如图 14-18 所示。这是一个标准的 RS-232C 电平转移电路，并连接到微处理器 UART 的 **RX** 和 **RT** 引脚。软件调试期间，此串口可用来将状态信息显示给上位计算机或终端。如果你非常有想象力，可以让上位计算机通过串口传送命令给火车控制器。如果你为机器人采用此设计，串口将会是非常有用的器件。然而，如果你要从外部计算机来进行控制，别忘了将H-bridge的**DIR**输入端连接到PIC，而不是直接连接到按钮。

下一章将要看到的是 AVR 系列处理器。这款处理器在 I/O 与性能方面与 PIC 处理器相当，不过它具有较高的处理能力以及比较丰富的结构。

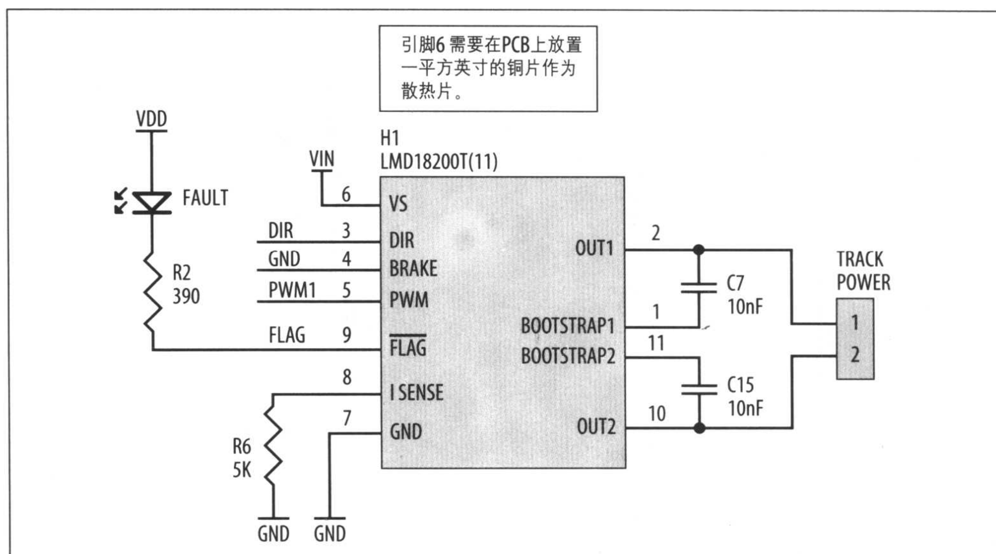


图 14-17：H-bridge

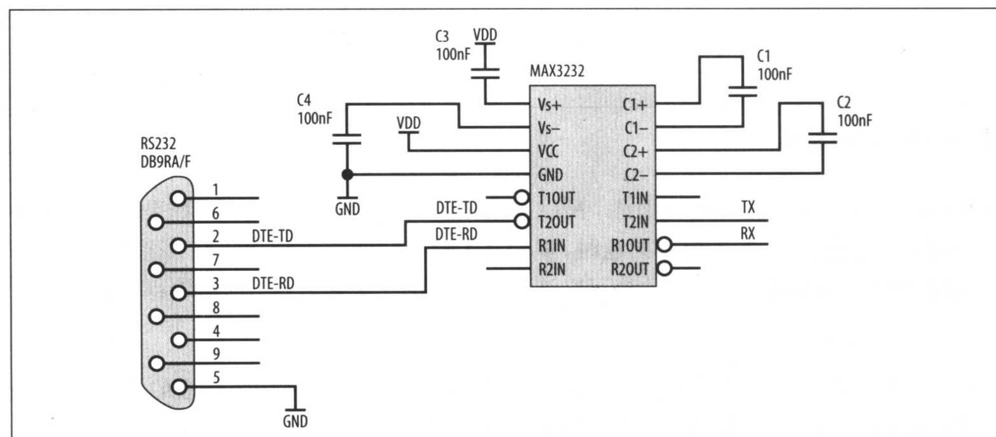


图 14-18：串口

第 15 章

AVR 微控制器

一个真正有用的引擎……

—— W.V. Awdrey

在这一章，我们将要介绍 Atmel 公司的 AVR 处理器。与 PIC 处理器一样，这一系列的处理器都是集成到单个芯片的独立的计算机。对于任何一种小型的控制或监视应用来说，AVR处理器都是理想的选择。它们包含有一组内置的片上外设，还可以在片外扩充附加功能。

与PIC一样，AVR处理器也采用RISC技术。根据我个人的经验，在这两种结构中，AVR的指令执行速度更快，可是为它们编写代码就不知道哪个更简单了。PIC与AVR一样使用的都是单周期执行的指令。不过我发现 AVR 有一个更加通用的内部结构，因而实际上使用它可以获得更高的吞吐量。如果让我给一个小规模嵌入式应用选择处理器的话，那么 AVR 将是我的首选。

在本章中，我们将通过设计一个小型的基于 AVR 的计算机系统来学习开发计算机硬件的基础知识。这个小型的计算机所使用的处理器为 AVR 系列的 ATtiny15 处理芯片。我们还会学习如何将代码下载到基于AVR的计算机中，以及如何在电路内再编程。之后，我们将会继续介绍一些更大、功能更强大的 AVR 处理器。

本章的后半部分主要讲述了如何使用地址总线、数据总线和控制总线来完成处理器与内存以及外设间的互连。这种接口方式也为大多数的处理器所采用，因而对AVR而言，可用的存储器和外设的种类是很多的。所以知道如何连接那些基于总线的设备，将能极大地扩充嵌入式计算机的应用。系统可以连接的设备多种多样，包括RAM、ROM（或闪存）、串行控制器、并行端口、磁盘控制器、声卡、网卡以及用以控制其他设备的主机。

大多数的微控制器都是完全独立的整体，并不向外“牵引”出总线。本章所要学习的

ATMEL公司的AT90S8515处理器，是AVR系列产品中唯一一种可以从外部访问其CPU总线的处理器。不过在学习之前，我们先来大体上看一下 AVR 的体系结构。

AVR 处理器的体系结构

AVR 处理器在挪威开发，并由 Atmel 公司负责生产。这是一款采用哈佛结构的 RISC 处理器，其设计的主要目的是加快指令的执行速度并减少系统的功耗。它有 32 个 8 位的通用寄存器（r0~r31），其中的 6 个可以复合成 3 个 16 位的索引寄存器（X、Y、Z），如图 15-1 所示。它总共有 118 条指令，故而可以提供多功能编程环境。

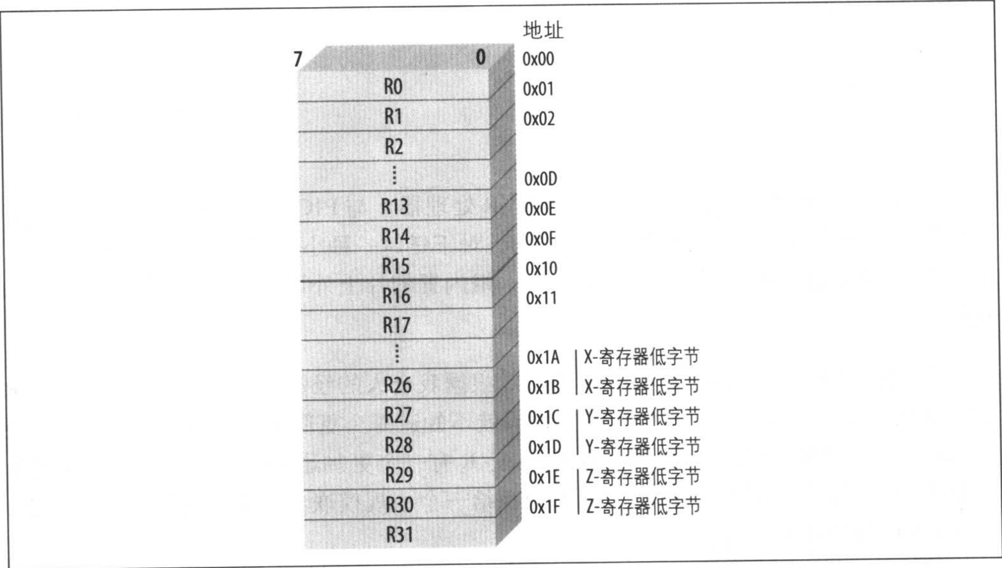


图 15-1：AVR 寄存器

在大多数 AVR 处理器中，堆栈存在于通用内存空间中，因此可以由指令直接操作，而且堆栈大小不受限制，而不像 PIC 那样，堆栈的大小已经固定（译注 1）。

AVR 拥有独立的程序空间和数据空间，最大可支持 8M 的地址空间。AT90S8515 AVR 处理器的内存映射如图 15-2 所示。

一直让 Atmel 公司引以为荣的是 AVR 处理器的吞吐量。该公司给出了如下一段示例 C 代码，这段代码可以在多种不同的处理器上编译运行：

译注 1：PIC 使用硬件堆栈，而 AVR 的堆栈在一定程度上也受制于 SRAM 的容量。

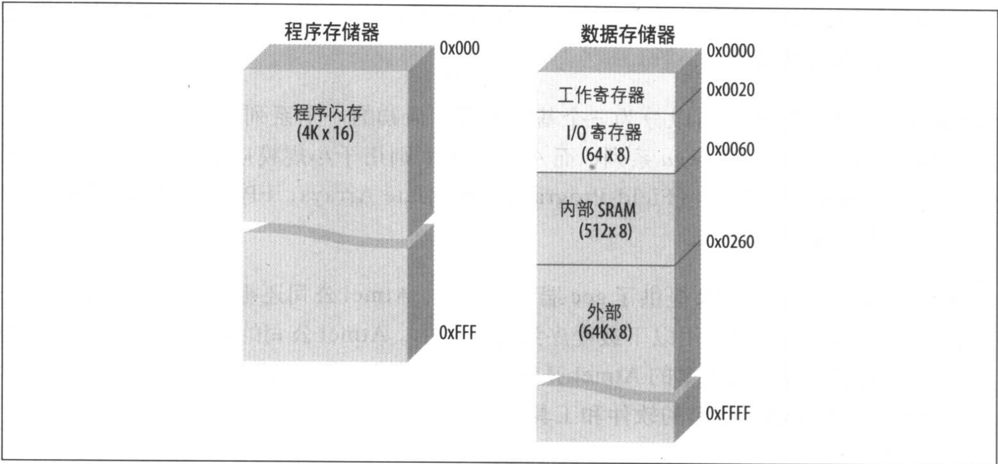


图 15-2：Atmel AT90S8515 内存映射模式

```
int max(int *array)
{
  char a;
  int maximum = -32768;

  for (a = 0; a < 16; a++)
    if (array[a] > maximum)
      maximum = array[a];
  return (maximum);
}
```

程序的运行结果很有趣（见表 15-1）。

表 15-1：几种处理器执行速度与效率的比较

处理器	编译代码的大小	执行时间（周期）
AVR	46	335
8051	112	9384
PIC16C74	87	2492
68HC11	57	5244

从上表中我们可以看出，以相同时钟频率运行时，AVR 的执行速度是 PIC16 的 7 倍，是 68HC11 的 15 倍，是 8051 的 28 倍之多！换句话说，如果 8051 要达到频率为 8MHz 的 AVR 的速度，那么它的时钟频率至少得是 224MHz。但有一点要说明，就是 Atmel 公司并没有说明他们在测试中使用的是哪一种编译器，而且对于选定的适当代码的编译效果

会有这样那样的差异。但从我个人的使用经验来看，AVR 确实能使你的程序执行得更快，而且代码也更紧凑。

AVR 体系结构的处理器可以分为三个基本系列。最初的一个系列是 *AT90xxxx*，对于复杂的场合，可以使用 *ATmega* 系列、而 *ATtiny* 系列则用于小规模应用。Atmel 也生产大规模的现场可编程门阵列（Field-Programmable Gate Arrays，FPGA），它包括了 AVR 核和大量的可编程逻辑门。

为了软件开发，还为 AVR 提供了 gcc 端口，同时，Atmel 公司还提供了一个汇编器、一个仿真器以及一套软件，用以下载程序到处理器中。Atmel 公司的开发软件可以在他们的网站上免费下载。低成本的 Atmel 开发系统是进行 AVR 开发的很好的工具，它为你开始 AVR 开发提供了所需的软件和工具。

下面我们将要讨论的 AVR 处理器是小型 ATtiny15、AT90S8535/AT90S4434 以及 AT90S8515。

ATtiny15 处理器

对很多简单的数字应用来说，小型微处理器较离散逻辑电路是一个更好的选择，因为前者能执行软件，所以它在执行某些任务时可以降低硬件的复杂度。下面我将介绍的是如何用一片 Atmel ATtiny15AVR 处理器构建一个小型的嵌入式计算机。你可以看到这个过程非常得简单。ATtiny15 有 512 字的闪存用以存储程序，但没有 RAM（想象一下，你是把以百兆计的应用程序装入PC机的）。与其他大的同类产品不同，ATtiny15 处理器完全依靠它的 32 个寄存器来存储中间变量。

由于没有 RAM 来配置堆栈空间，因而 ATtiny15 采用了专用硬件堆栈，但堆栈深度只有 3 个输入项，而且为子程序调用和中断所共享。（可以想象如果嵌套超过三层，结果将会如何！）程序计数器为 9 位（用以寻址 512 个字的程序存储空间），因而堆栈也是 9 位。和较大的 AVR 处理器的不同之处还有：只有两个寄存器（R30 和 R31）可以联合组成一个 16 位的变址寄存器（称为 Z）。

处理器还包括 64 字节的 EEPROM（用于存放系统参数）、高达 5 个的通用 I/O 引脚、8 个内部和外部中断源、两个 8 位定时器 / 计数器、一个 4 通道的 10 位 A/D 转换器以及一个模拟比较器，而且处理器还能进行电路内的再编程。它的封装带 8 个小引脚，可获得高达 8MIPS 的吞吐量。对于它的大部分特性本章先不介绍，在后面介绍 I/O 特性时会一并讨论。目前我们将专注于如何才能用这样一个元件来完成简单的数字控制。

使用一个像 ATtiny15 那样的小型控制器非常简单，因为它只需很少的外部支持就可以实现自身的操作。图 15-3 给出了非常简单的一个 ATtiny15。

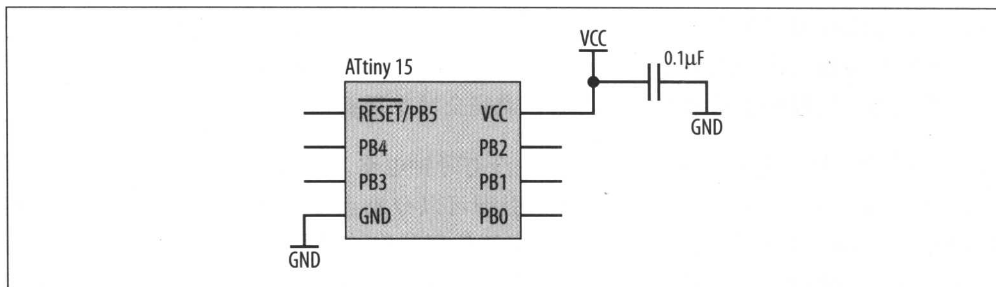


图 15-3：一个简单的 AVR 计算机

让我们快速地看一下 ATtiny15 的设计。**VCC** 用以连接电源，电源的电压可以介于 2.7V~5.5V 之间。**VCC** 使用一个 0.1μF 的电容接地，用以退耦。五个引脚 **PB0~PB4** 可以用作数据的输入或输出。它们可以完成多种功能：读取开关的状态，打开或关闭外部设备，产生信号波形用以控制小型电机，甚至还可以作为一些简单外设的接口。这些数据线（**PB0~PB4**）还可以用来连接到任何处理器负责监控的设备上。稍后我们将在本章给出几个相关的例子。

最后，还有一个输入端（**RESET**）没有任何连接，这对其他任何处理器来说是不可思议的。许多处理器都需要一个上电复位电路（Power-On Reset，POR），以便在程序执行时使系统初始化。还有一些处理器具有一个内部上电复位电路，因而不需要外部电路支持，但这样的处理器仍会有一个复位输入，使用户或外部系统可以手动复位。一般情况下复位端仍需要一个上拉电阻，以使其处于非激活状态。但ATtiny15处理器没有这个需要，因为它内置了一个上电复位装置和一个上拉电阻。因此和其他大部分（也许是全部）的处理器不同，**RESET** 端在 ATtiny15 上是处于非连接状态的。事实上，如果 ATtiny15 不需要一个外部复位电路，那么它的 **RESET** 引脚也可以作为通用输入端（**PB5**）来使用。但必须注意的是：由于 **RESET/PB5** 没有通常的输入电压保护机制（用来防止高于正常值的电压输入），因为在程序烧录器（program burner）下载软件时，**PB5** 端的电压将会升至 +12V。因此，当你在使用 **PB5** 引脚时，一定要注意不要使输入电压超出 VCC 1V 以上。如果不能做到这一点，处理器将会处于软件下载模式，这样绝对会使你的嵌入式计算机完全崩溃。

AVR处理器（PIC也是如此）还包括一个内部电路，叫作节电检测器（brownout detector，BOD）。BOD 能检测出处理器电源供应的微小波动，这个小小的波动有时会影响指令的执行。一旦检测到这种波动，BOD 就产生复位指令，使计算机重启。ATtiny15 处理器还有另外一个复位发生装置，叫作看门狗（watchdog），用以在软件崩溃时重新启动计算机。看门狗是一个小的定时器，其作用是当超时发生时重启计算机。正常操作时，软件有规律地重置看门狗。它们的工作原则是“在你复位我之前，我就复位你”。如果软

件崩溃，而监视器又未清零，这时超时就发生了，计算机就被重启。带有看门狗的处理器能使软件有能力区别上电复位和看门狗复位。对于看门狗复位，就可以从内存中恢复系统的状态，使程序能够继续执行，而无需完全地重新初始化。

ATtiny15 的设计还有一个独特之处就是它没有时钟电路。ATtiny15 芯片可以有一个外部晶体电路（ATtiny15 的 **PB3**、**PB4** 引脚可以分别当作晶体的输入端 **XTAL1** 和 **XTAL2**）。但在我们的设计中没有也没有必要使用晶体，因为处理器自身有一个完全内置的频率为 1.6MHz 的振荡器（本例中为 RC 振荡器），所以就没有必要为时钟再额外添加一个外部元件。但问题在于振荡器是不太稳定的，它的频率可能会随着温度的改变而变化（ATtiny15振荡器的工作频率可在80MHz~1.6MHz之间变化！）。通常情况下，RC 振荡器并不适合那些对时间要求比较高的应用，对于那些应用就需要用外部晶体来代替。但如果你的ATtiny15只是做一些简单的控制功能，而且时序也不是个问题，就可以通过使用内部RC振荡器来降低硬件复杂度。Atmel 公司还为 ATtiny15 提供了一个 8 位的校准寄存器，所以你能够随时调整内部振荡器，使之精确度更高。

至此我们已经看了ATtiny15机器的基本设计，总得来说它是一种低价格、多功能的小型计算机，也并不需要对内部再作任何工作。设计时需要做的努力就是确保计算机能够同与其接口相连的 I/O 设备很好地工作。如果你不打算用电池组给你的系统供电，那么电容是可以选择的！只有处理器才是不可缺少的(或许在此之前你还以为构建一个计算机系统是多么困难的一件事呢)。

这就是基本的 AVR 计算机硬件，只需极少量的元件。我们马上就会看到如何快捷地为你的硬件下载软件。

从上面所介绍的 ATtiny15 系统的基本情况可以看出，它与同类型的 PIC12C508 系统没有太大的不同。它们间的真正区别在于其内部结构（和指令集）、对操作电压的敏感度以及接口功能等几方面。正如你所看到的那样，把这类小元件加到你的嵌入式系统并不是一件很困难的工作。

到目前为止，我们介绍的计算机都没有连接任何外部设备。下面就开始一些简单的工作，为AVR添加一个LED。这种基本的连接技术也同样适用于所有带可编程I/O线的微控制器。

添加一个状态 LED

当有电流通过时，LED便会发光。由于二极管具有单向导电性，所以只有当电流从阳极（正极）流向阴极（负极）时，二极管才能工作。在二极管的电路符号中，阴极用水平横线表示，阳极用三角形表示。

图 15-4 所示的是状态 LED 的电路图，它使用微控制器上的一根 I/O 线来控制 LED 的开关。当 I/O 线输入低电平时，LED 启动；输入高电平时，LED 闭合。图中电阻 R 的作用是防止太强的电流流入 I/O 线路。

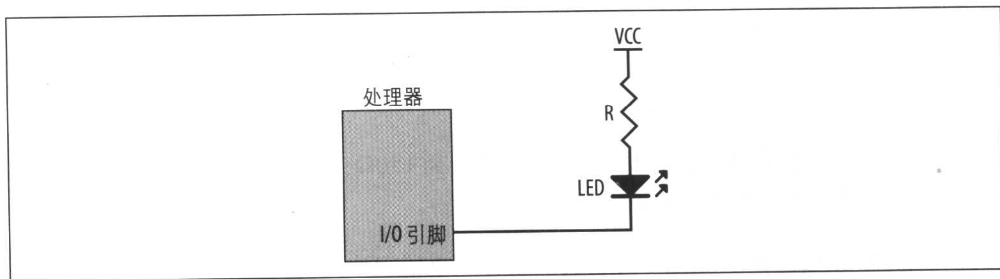


图 15-4：状态 LED

当有电流通过时（此时 LED 发光），LED 有一个正向电压降，表明阴极的电压要低于阳极的。不同的LED电压降的幅度有所不同，可以通过查找相应的技术手册来获得你所使用的 LED 电压降的数值。

当处理器的工作电压为 5V 时，ATtiny15 的 I/O 引脚输出的低压为 0.6V；当处理器的工作电压为 3V 时，输出的低压则为 0.5V。假定我们使用的是电压为 5V（为了本例计算方便）的电源 **VCC**，LED的正向电压降为 1.6V。现在，要将LED 阴极输入电压置为 0.6V，我们可以这样计算：电源 **VCC**（5V）与 LED 阴极间的电压为 4.4V，如果 LED 的电压降为 1.6V，那么电阻上的电压降应该是 2.8V（由 5V－1.6V－0.6V=2.8V 可得）。

从技术手册可知，如果处理器的工作电压为 5V，则 AVR 的数字引脚上的电流可达 20mA。显然，我们必须限制电流的流量，引入电阻就是出于这个目的。如果电阻的电压降为 2.8V（上面已计算过），流入的电流为 20mA，那么由欧姆定律可知，我们所需的电阻的大小为：

```
R=V/I
 =2.8V/20mA
 =140Ω
```

我们就选择与这个值最接近的可用电阻，即 150Ω（使用 150Ω 的电阻将会使电流为 18.6mA，这正合适）。

警告： 当工作电压为 5V 时，AVR 的每个引脚可以接受的电流为 20mA。不过随着工作电压的减小，可接受的电流也随之减少。当工作电压为 2.7V 时，AVR 可接受的电流就只有 10mA。所以，在使用前仔细阅读技术手册是很有必要的。

另一个问题是：电阻会消耗掉多少电能？换句话说，当电压降为2.8V时，电阻会消耗掉多少能量？这个问题很关键：如果通过电阻的电流过大，那么电阻就有可能被烧坏。所以，在选择电阻时要选择一个额定功率大于所需值的电阻。功率可由电压和电流的乘积计算出：

```
P = V * I
  = 2.8V * 20mA
  = 0.056 Watts = 56mW
```

这个值是微不足道的，所以我们可以选择电阻值为150Ω、功率为0.0625W的电阻（因为0.0625W是目前可用电阻中额定功率的最小值）。

因此，当I/O线由过高电压驱动时，会发生什么事情呢？此时（工作电压为5V）AVR的I/O引脚的输出电压至少为4.3V，则此时VCC与LED阴极间的电压降只能是0. 7V（或更少）。而LED的正向电压降（即阈值电压）为1.6V，故没有足够的电压使其导通。

在这种情况下，只需使用简单的处理器的数字输出，就可以控制LED的闭合与导通了。我们还知道了如何计算电压和电流，这对于设计的每一方面都非常重要。否则，你将会设计生产出很糟糕的机器。而且更可怕的是，这样也许会烧毁芯片，使空气中弥漫着晶体硅焦臭的味道。

下面我们要看到的是如何使用AVR的数字输出端去控制LED。这种方法对于那些工作电流低于20mA的设备同样有效。事实上，小功率的元件（如传感器），都可以使用AVR的输出端来对电源进行直接控制，就像我们直接控制LED的电源那样。在那些使用电池组供电的应用中，这也是一个很有用的技术，因为它可以降低系统的总功耗。

切换模拟信号

我们还可以使用处理器的数字I/O线路来控制系统内的模拟信号流。比如，我们的嵌入式计算机有可能被集成到一个音频系统中，用以控制不同的音频源之间的切换。为了完成这一任务，我们还需要为每一信道准备一个开关（Switch），如MAX4626。这种微小的元件（以贴片技术所设计的MAX4626只有一粒米那么大）需要单独的工作电压（其值介于1.8V~5.5V之间）。它还有一个内置的过载保护电路，以免设备在短路时被损坏。图15-5所示的是MAX4626与ATtiny15的连接电路图。AVR输出端（**PB2**）为高电平时，MAX4626将处于工作状态，并连通**NO**和**COM**端。将**PB2**置为低电平会使连接中断，通过这种方式，MAX4626便可以在软件的控制之下完成输入端与输出端的连接。

问题是：MAX4626与AVR兼容吗？当工作电压为5V时，MAX4626的输入端（引脚4，**IN**）需要的逻辑低电平输入要小于0.8V，逻辑高电平输入至少为2.4V。而AVR的逻辑低电平输出低于（或等于）0.6V，逻辑高电平输出高于（或等于）4.3V。因而，AVR

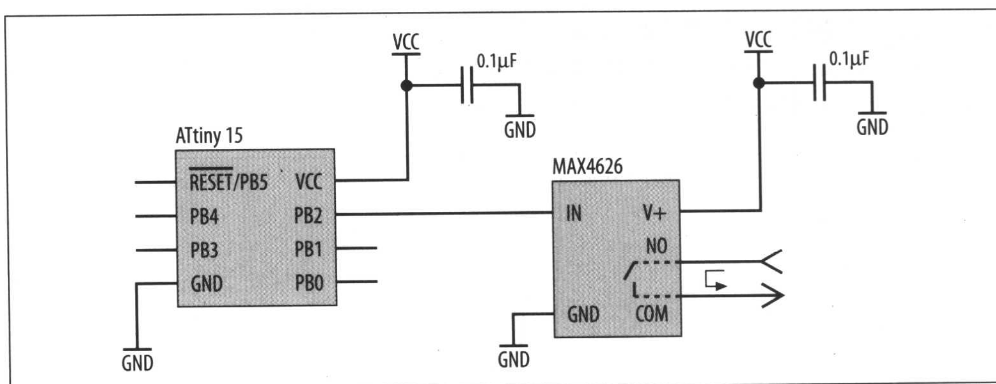


图 15-5：模拟信号的切换

的数字输出端电压完全符合MAX4626的要求。再来看电流，MAX4626接收或发送的电流只有 1μA，所以对 AVR 来说，这不是什么问题。

即使 MAX4626 不合适，MAXIM 公司或其他制造商也还生产了多种不同特性的类似产品，相信总会有适合你需要的产品。

图 15-6 所示的电路原理图给出了一个连接 **PB3** 的按钮，此处 **PB3** 作为数字输入端。不过，对于这个简单的输入电路，有两个有趣的问题要指出。第一，并没有外部上拉电阻连接 **PB3**。而通常情况下这种电路是需要一个上拉电阻在当按钮放开时（即未被按下）将输入置为一个已知状态。除了按钮闭合或输入端接地时，上拉电阻都将输入端置为高电平。本例中没有上拉电阻的原因是：在 AVR 内部已包含有一个上拉电阻，并且可以由软件控制是否选择上拉电阻的功能。

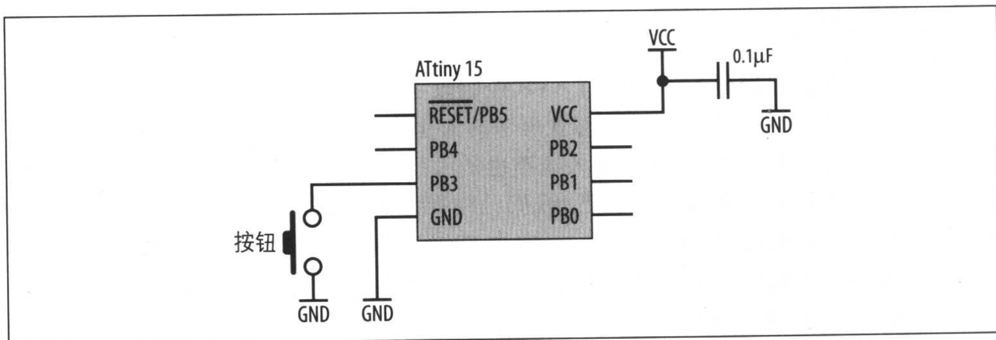


图 15-6：按钮输入

第二个需要说明的有趣的事情是，在按钮与输入端之间没有连接去抖动电路。当按下任何一种机械开关（包括键盘的按键）时，它都是一个小的感应器。其结果是会在信号线上产生一个迅速衰减的振铃振荡，如图 15-7 所示。

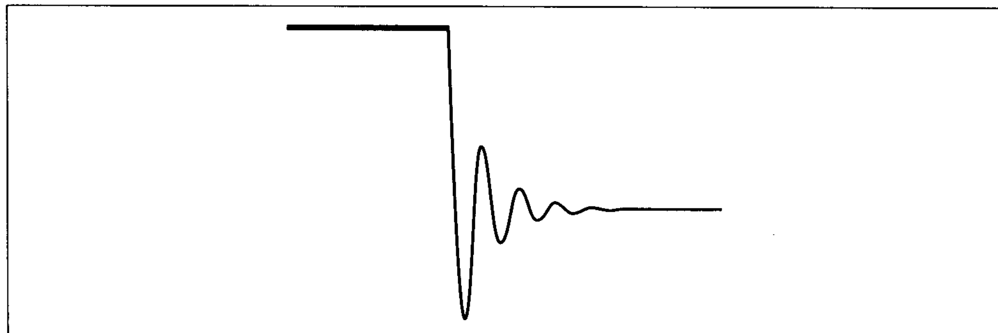


图 15-7：信号的抖动

因此，可以看到当按钮被按下时，状态并不仅仅改变一次，却像是用户在快速连续地敲击按钮，于是负责响应这些输入变化的软件会记录多次脉冲信号，并不是像用户所希望的单一信号。所以，从信号中消除那些瞬时的过渡状态是很重要的，这个过程被称作去抖动。目前，有很多种不同的电路可用来消除这些振荡信号。但问题是，你并不是非要用这些电路不可！当一个用户按下按钮时，他通常不立即放开，而是持续至少半秒钟，直至振荡信号完全消失。所以这一问题也可以用软件的方法来解决。当软件第一次记录的输入电平信号为低，而且如果隔几百微秒后采样所获得的信号（当然，可能不止采样一次）也为低时，那么软件就可以断定这是一次有效的按钮输入。之后，软件会“重置”输入端，以等待下一次按键。但如果按钮与中断电路或者复位端相连的话，那么去抖动硬件就很有必要了。

到目前为止，我们已经学习了如何用 AVR 控制数字输出，以及如何处理一些简单的数字输入。在观察上两个电路图时，喜欢思考的人也许会问：难道处理器是必不可少的吗？将按钮直接连到MAX4626的输入端不也是可行的吗？那么处理器在这里到底起的是什么作用呢？现在我来解释一下吧。我们已经看到处理器的一个作用，即取代输入端的去抖动电路。处理器另外的作用是：由于处理器有内置的存储器，能够执行软件，所以可以用来记录系统的状态（和模式）、监控多种信号的输入、为输出端提供复杂的控制序列。简而言之，使用微处理器可以减小系统的硬件复杂度，增强系统的功能。所以，它们是非常有用的工具。通过使用更先进的处理器、更多种类的 I/O，嵌入式计算机能够提供的应用和功能也就越来越强大！

代码的下载

AVR 使用内部闪存来存储程序，还可以实现电路内编程。如果使用的是插座元件(socketed component)，则也可以在电路外编程。AVR处理器通过芯片上的串行外设接口SPI来实现再编程。甚至像ATtiny15这样的处理器自身也不包含SPI接口，在进行重复编程时需要配合SPI端口使用。当处于编程状态时，芯片的引脚**PB0**、**PB1**以及**PB2**分别连接到SPI的功能端（**MOSI**、**MISO**和**SCK**）。

VCC电压可以由用于下载代码的编程器提供。当处于编程状态时，**VCC**必须是5V。如果嵌入式系统的内部电源能提供5V的电压，那么编程器上的**VCC**端可以不用。但如果嵌入式系统电压不是5V，则编程器的**VCC**端一定要用，同时要禁用系统的内部电源。复位键$\overline{\textbf{RESET}}$的作用也很重要。当$\overline{\textbf{RESET}}$启动时（置为低电平），系统开始编程，此时，处理器内部的CPU被禁用，因而可以访问内部存储器。引脚**PB0**、**PB1**和**PB2**的功能也随之改变，以连接SPI接口。然后，开发软件通过SPI接口将某一代码串发送出去，用以“解锁”程序存储器和启动下载软件。一旦程序下载开始，写指令序列就会被执行，软件（以及相关设置）就被逐字节地下载。Atmel公司的软件可以完成这一整个过程，所以正常情况下你不必关心具体的实现细节。但如果你需要“手动地”完成这一过程，比如从其他类型的主机进行操作，则可以从Atmel的技术手册上查询相关的实现协议。

Atmel开发系统还带有一个专用的适配器电缆，它可以插入到开发面板上，这样你就可以使用PC机的并行端口对微处理器进行再编程了。如果在电路内安装了合适的连接器，你也可以在嵌入式系统中使用同样的编程电缆。根据开发面板的特点，你可以从两种可能的连接器中选择一种实现电路内编程。图15-8所示的是它们的引脚图。**VTG**用于为目标系统提供输入电压。但如果目标系统有自己的电源且输出电压可以达到编程的要求(+5V）的话，那么**VTG**引脚也可以不用。在某些Atmel应用说明中，引脚3通常没有连接。不过，在一些开发系统中引脚3通常被用来驱动LED(用以表明程序正在下载中)。

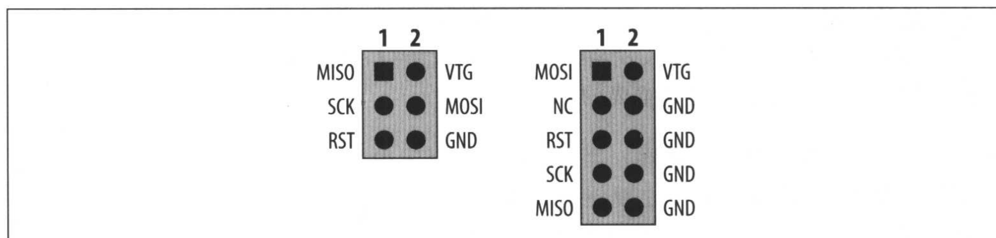


图 15-8：电路内编程连接器

使得计算机支持电路内编程的电路图如图15-9所示。需要指出的是，连接器的**MOSI**端将连接到处理器的**MISO**端，同样，处理器的**MOSI**端连接到连接器的**MISO**端。这样做的原因在于，在下载过程中处理器一直处于从机（slave）地位而非主机（master）地位。

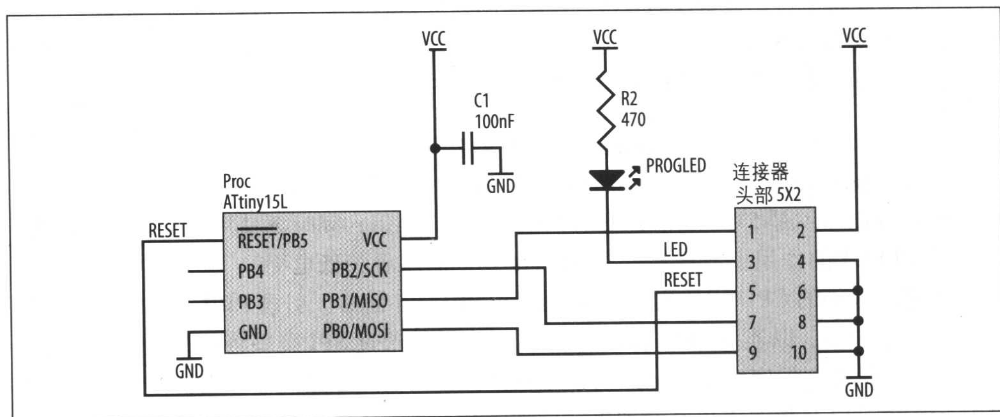


图 15-9：电路内编程

连接器的类型可以看作*IDC*的头部，电缆可以提供编程所需的所有命令，其中也包括驱动编程LED指示灯信号的指令。当不用于编程时，连接器还可以作为嵌入式系统的一个简单的I/O连接器，用来访问数字信号。因而一种硬件可以起到两个作用。

然而，有一点非常重要：如果你使用**PB0**、**PB1**或**PB2**连接了系统内部的元件，那么必须要小心，在下载代码时不要对它们造成不利的影响。比如，一个带有MAX4626的电路使用了**PB2**引脚作为控制输入端。在代码下载的过程中，**PB2**会作为一个时钟信号（SCK）。因而，在代码下载到处理器的过程中，MAX4626会一直在开与关之间来回快速地切换。如果此时MAX4626在控制某种设备，那么该设备同样会不停地打开、闭合，这就有可能带来灾难性的后果。从另一方面来说，如果你的系统中还有其他的元件，那么在下载的过程中，一定不要试图将其连接到**PB0**、**PB1**或**PB2**引脚上。如果这样做了，轻则下载中断，重则会损坏嵌入式系统和编程器！所以在编程时，充分考虑代码下载对系统内其他组件的影响是极其重要的。

那么我们该如何解决这一问题呢？当然，我们可以使用**PB3**去控制MAX4626，因为它不参与编程。不过如果我们一定要使用**PB2**的话，那么我们可以在处理器与MAX4626之间加入一个缓冲器（可由**RESET**控制）。当**RESET**为低电平时（处于编程状态），缓冲器失效，则MAX4626被隔离。还有一种方法就是使用采用DIP封装技术的同种类型的处理器。它通过插槽安装，需要重新编程时，可将其物理地取下。如果你使用的处理

器采用的是贴片封装技术，它可以被安装到一个小的 PCB 面板上，而该 PCB 面板又可以插入到嵌入式系统中（就像你的PC机上的SIMM内存条那样），那么在编程时也可以将其取下。方法远不止这些，哪种方法最好要由应用决定。

有些AVR处理器（不包括ATtiny15）具有使用 SPM（Store Program Memory）指令来修改自身程序存储器的功能。使用这样的处理器，软件可以通过处理器的串口来下载新的代码。为了完成这一过程，你的处理器需要预装载引导装入程序。正常情况下，在初始化过程中，需要给所有的处理器都加载引导装入程序（以及你的应用软件的1.0版本）。此后自我编程（self-programming）机制才可以用来更新你的应用软件。为方便起见，程序存储器被分为两部分：引导区和应用程序区。内存空间被划分为128或是256字节大小的页（具体大小由处理器类型决定）。一次只对内存的一个页进行擦除和再编程。在编程过程中，Z寄存器作为指针用以存放页地址，而r1和r0寄存器则一起用来存放待编程的数据字。Atmel 应用参考（《AVR109：自编程》，可以从该公司网站上查阅）给出了引导装入程序的示例源代码，并给出了详细的说明。

无论你使用的是何种类型的处理器，芯片的制造商都会给出相应的技术参数，用以说明如何将代码下载到处理器中。

更强大的 AVR 处理器

到目前为止，我们所介绍的都只是小型的 AVR 处理器，它们的功能都很有限。由于嵌入式系统中可以使用各种形式的I/O接口，因此我们需要功能更多的处理器。前面已经详细介绍了 ATtiny15 处理器，现在我们将要看到更为复杂的处理器。

首先要介绍的是 Atmel AT90S8535。这是一款中型的 AVR 处理器，内置有大量的 I/O 端口。除了内置的I/O端口，它还有各种各样的端口，如：UART、SPI、定时器模拟输出的 8 个通道、模拟比较器及用于参数存储的内部 EEPROM。此处理器有 512 字节的内部RAM和用来存储程序的8K闪存。它的一个更小的姊妹产品 AT90S4434，除程序存储空间更小（4K的存储程序空间以及256字节大小的RAM）外，其他方面几乎完全相同。但如果只从电器特性的角度来看，AT90S8535 和 AT90S4434 处理器没有什么差别。

图15-10所示的是基于AT90S8535芯片设计的计算机的电路图，图中没有添加任何额外的设备。可以看到，除了有更多的引脚之外，别的方面它与 ATtiny15 没有什么大的不同。在 **RESET** 端有一个 10kΩ 的外部上拉电阻。处理器还有一个外部晶体（X1），这需要有两个小的旁路电容：C1 和 C2。处理器还有 4 个电源引脚，每个引脚都连接有一个

100nF的陶瓷电容用以退耦。这些电源输入端中有一个**AVCC**，是用来给芯片内的模拟部件供电的，一个100Ω的电阻R2将其与数字电源输入端隔离。它为模拟部件与开关噪音之间提供了一个小的屏障，而这些开关噪音也许来自数字电路。和ATtiny15芯片一样，其余的引脚都作为通用的I/O端口。不过与ATtiny15的不用之处在于，这些引脚都具有两种不同的功能。它们可以在软件的控制下重新设置，用以实现另一种I/O功能。处理器的技术手册上给出了如何在软件的控制下，去设置这些功能的全部信息。

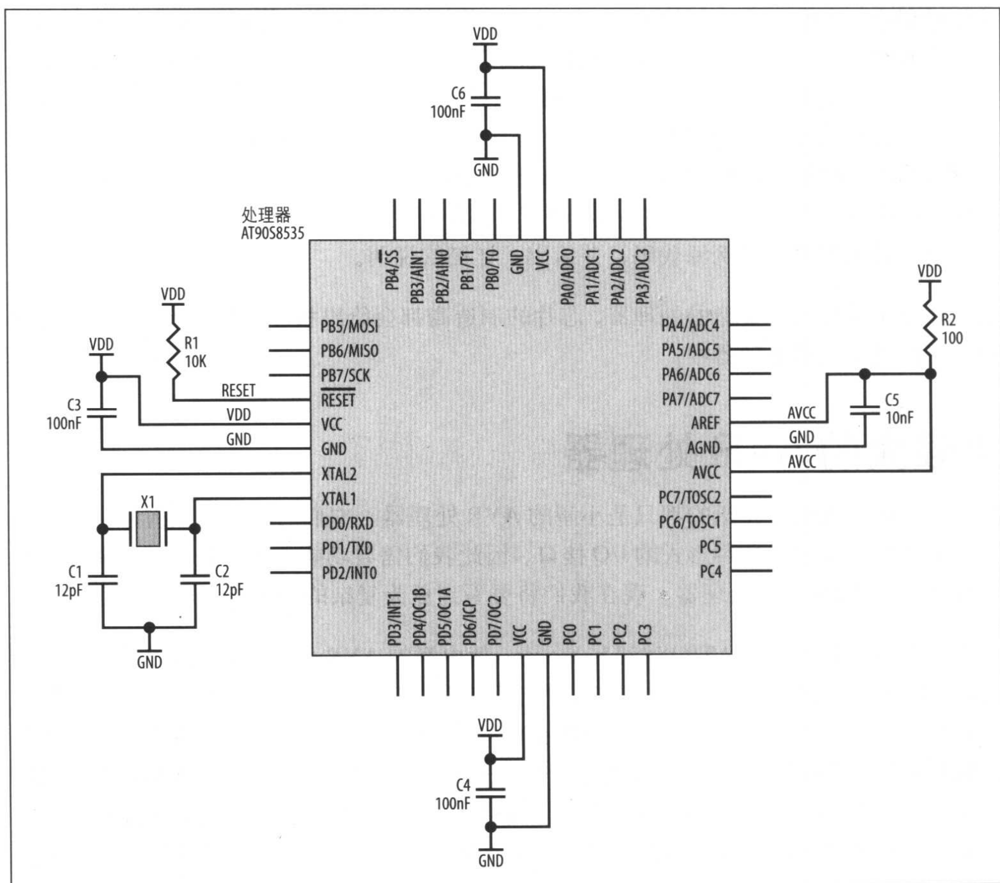


图 15-10：AT90S8535 处理器及其支持元件

对大部分的AVR处理器而言，基本的AVR设计思想都是一样的。引脚或许有所不同，但其所需的基本支持都差不多。无论如何，查阅合适的技术手册是必要的，它将给出你所用处理器的实现细节。

基于 AVR 的数据记录器

前一章中，我们看到过如何设计一个基于 PIC16F873A 的数据记录器。设计一个基于 AVR 的数据记录器并没有什么太大的不同。图 15-11 所示为此设计的基本电路图。

如同PIC处理器一样，Atmel的90S4434 AVR处理器也是经由简单的SPI来连接串行数据闪存芯片。电路图中AVR的部分跟前面所看到的没有什么两样。这对SPI之类的简单接口而言是一件好事。它们各自形成子系统模块，就像可以“扣在一起”的积木一样。首先是设计的核心，然后再加上你所需的外部器件。电路图中还可以看到电源供给的退耦电容、连接到处理器的晶体振荡器，以及连接到 **RESET** 的上拉电阻。41 引脚（**PB1**）被当作闪存的“手动”（由处理器来控制）复位输入端。

模拟输入 **ADC0:ADC7** 可以连接到一个IDC连接头，以允许对外部进行采样，或是直接连到传感器上，正如我们在基于 PIC 的数据记录器中所看到的那样。串口信号 RXD 和 TXD 被连接到 MAX3233，这点也与 PIC 的设计一样。

总线接口

在这一节要介绍的是如何通过连接基于总线的存储器和外设来扩充处理器的功能。尽管不同的处理器体系结构具有不同的信号和时序，但是只要你掌握了其中的一种，其基本原则就普遍适用。由于绝大多数的小型微控制器都没有总线，所以我们的选择有限。我们将要看到的是 AT90S8515，它是唯一一个具有外部总线的 AVR 处理器。在 PIC 世界里，则是 PIC17C44 可以基于总线进行连接。

AT90S8515 存储器周期

存储器周期（也称机器周期、处理器周期）定义为处理器从发起存储器（或外设）访问，经过数据传输，再到访问终止的时间。存储器周期由处理器产生，通常是一个固定的时间段（或其倍数），可能是若干个（处理器）时钟周期。

存储器周期通常分为两类：读周期和写周期。对存储器和外设而言，当数据被选中（且读或写周期被确认）后，要求数据必须持续一段时间，这就给系统设计者提出了要求。胶合层逻辑（介于处理器和外设之间的接口逻辑电路）必须在一定的时间内执行自身的功能，如选择正在访问的外设。设置的时间必须要满足胶合层逻辑的时间要求，否则计算机将无法工作。胶合层逻辑电路监测来自处理器的访问地址并选择一个设备作为地址解码器。下面我们就会更详细地介绍地址译码器。

图 15-11：将一个 2MB 的数据闪存连接到 AT90S4434

在计算机的设计中，对时序的处理也许是最为关键的一个环节。例如，如果一个给定的处理器有一个 150ns 的时钟周期，而存储设备需要其中的 120ns 来完成发起访问和数据

传输工作，那么时钟周期最初的30ns则留给胶合层逻辑电路，用以处理处理器信号。一个74LS系列的TTL门电路典型的传输延迟为10ns。因此，在本例中可以使用两个以上的74LS门电路来实现地址解码电路，以便满足时间的要求。

同步处理器的存储器周期大小固定，处理器时序信号与时钟直接相连。还有一个假定，即系统内的所有设备都能被访问，而且能在给定的存储周期内作出响应。如果系统中某一设备的响应时间大于存储器周期所允许的时间，就需要增加逻辑电路来暂停处理器的访问，以使慢速设备有足够的响应时间。处于暂停状态时的时钟周期都称为等待状态。一旦慢速设备准备就绪，处理器再由逻辑电路释放，继续执行存储器周期。这种暂停处理器来同步慢速设备的方法称为插入等待状态。使处理器处于等待状态的电路就被称为等待状态产生器，它可以由一组触发器构成（触发器的功能是一个简单的计数器）。产生器由处理器的输出激活，以表明存储器周期的开始，并且在每个存储周期结束时复位，从而返回已知状态（还有一些处理器内置有可编程的等待状态产生器）。

而对于异步处理器来说（如68000），在给定的几个时钟周期内，它不会终止对存储器的访问。相反，它等待来自外设或辅助逻辑电路的传输确认，以判断它所访问的外设是否有足够的时间在存储周期内完成操作。换句话说，处理器能够在存储周期内自动插入若干等待状态，直到待访问的设备准备就绪。如果处理器没有收到确认，就会无限期地等待。因而，很多计算机系统在使用异步处理器时，都会用一个附加的逻辑电路来协同工作。该逻辑电路的功能是：当一个存储周期太长（即有可能不能终止）时，它就将处理器重启。另外，异步处理器还能作为同步处理器使用，只要把确认线路固定为激活状态即可。然后它假定所有的设备都能与之同步，这就是所谓的无等待状态运行。

绝大部分的微控制器都是同步的，而大多数大型的处理器则都是异步的。AT90S8515是一个同步处理器，内部有一个等待状态产生器，可以插入单个等待状态。

总线信号

图15-12所示是带有很少配套组件的AT90S8515处理器芯片。AT90S8515用来和外界接口的总线包括：一个地址总线、一个数据总线以及一个控制总线。由于处理器的引脚数目有限，所以这些总线通常会与数字I/O端口（端口A和B）共享引脚。控制寄存器中的一个比特位被用来区分引脚是用于I/O还是连接总线。现在一个16位的地址总线加上一个8位的数据总线一共是24位，但端口A与端口B却只有16位。那么处理器是如何使24位的总线来适合16位的端口呢？采用的方法很简单，就是将地址总线的低8位与数据总线复用。当存储访问开始时，端口A输出地址位A0~A7。另外，处理器还提供了一个控制线ALE（Address Latch Enable，地址锁存使能）用来控制锁存器，如74HCT573（如图15-12中右侧所示）。当**ALE**为低电平时，锁存器获得并保持低8位地址。同时，端口B输出高8位地址A8~A15。这些地址位在整个存储访问过程中都将保

持有效。一旦锁存器将低位地址锁存，那么端口A将成为处理器与外部设备之间的数据传输通道（数据总线）。图15-12所示的电路中还包括晶体电路、系统内编程端口、为处理器电源退耦的电容以及用于其他重要信号的网路标识。

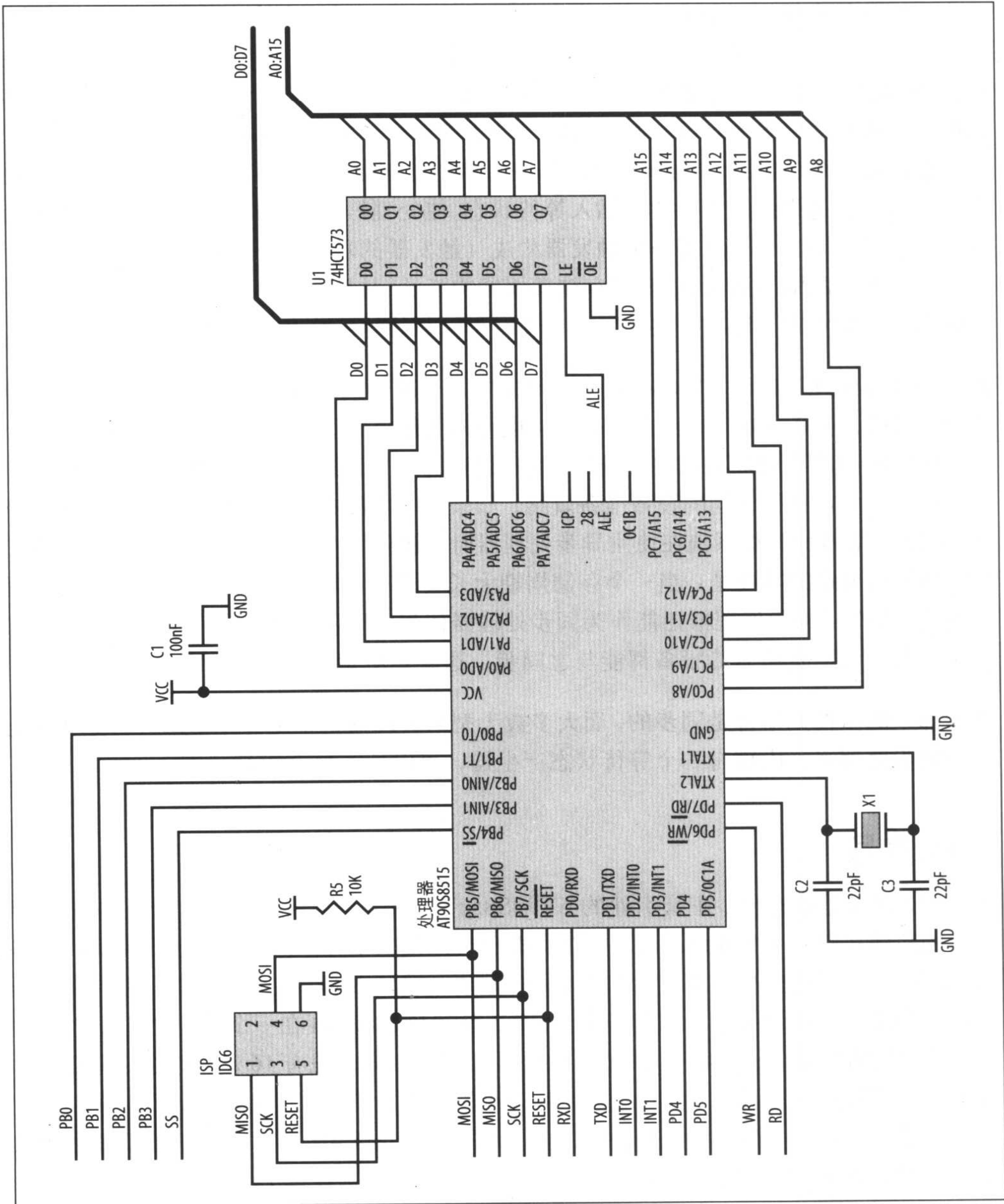


图15-12：地址总线的多路复用

图15-13所示为AT90S8515的时序图。只有当处理器的等待状态产生器被激活时，周期T3才存在。

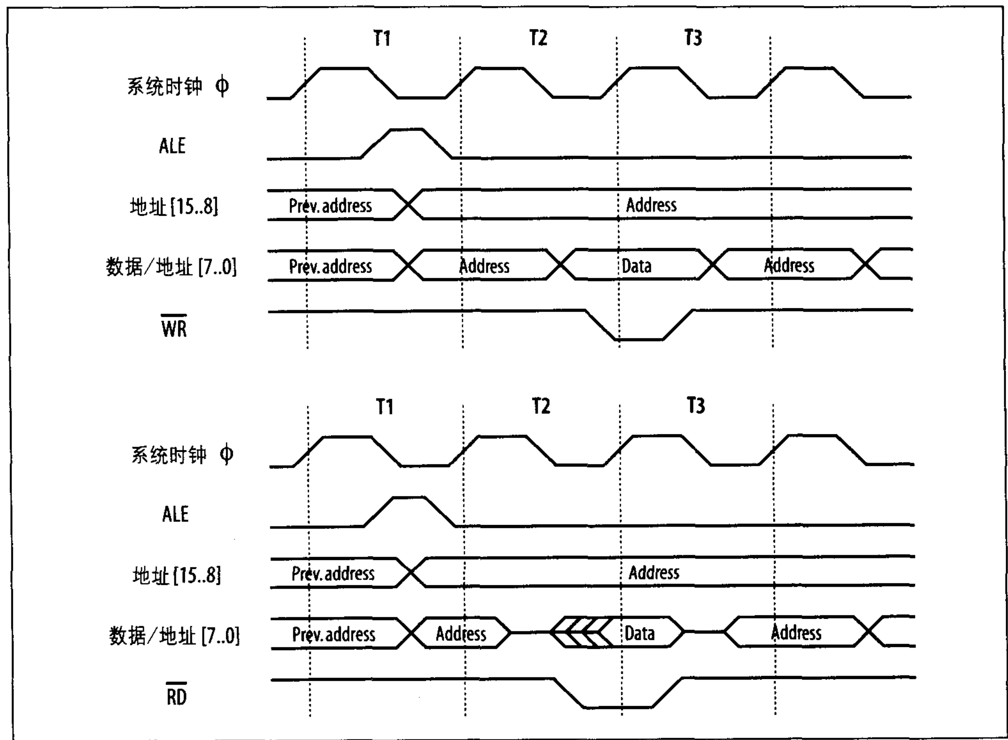


图15-13：AT90S8515存储器周期

现在，让我们更详细地看一下这些信号（在后面你还将看到如何应用这些信号，但现在我们要做的只是“浏览”一下信号的时序图）。时序信息的索引可以查询Atmel公司网站上的技术手册得到。图15-14所示的即是Atmel公司技术手册上的时序信息，并且具有完备的时序参考说明。

参考信息可以通过查找处理器的技术手册中相应的表格而得到（见表15-2）。

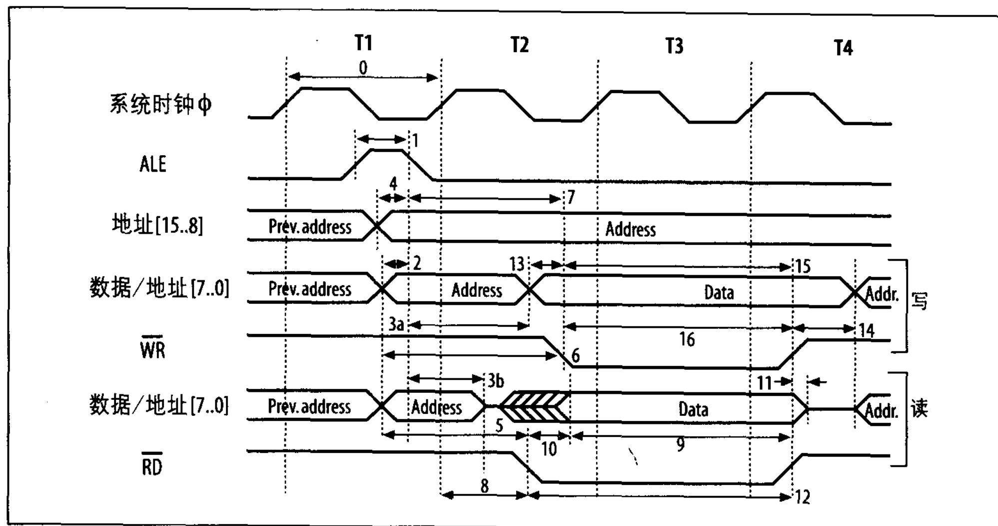


图 15-14：带有时序参数的 AT90S8515 存储器周期

表 15-2：时序参数表

标号	符号	参数	8MHz 振荡器 最小值	8MHz 振荡器 最大值	可变振荡器 最小值	可变振荡器 最大值	单位
0	$1/t_{CLCL}$	振荡频率			0.0	8.0	MHz
1	t_{LHLL}	ALE 脉冲宽度	32.5		$0.5t_{CLCL}$ −30.0		ns
2	t_{AVLL}	地址 A 有效直到 ALE 变为低电平	22.5		$0.5t_{CLCL}$ −40.0		ns
3a	$t_{LLAX...ST}$	ALE 为低电平后地址保持，ST/STD/STS 指令	67.5		$0.5t_{CLCL}$ +50.0		ns
3b	$t_{LLAX...LD}$	ALE 为低电平后地址保持，LD/LDD/LDS 指令	15.0		15.0		ns
4	t_{AVLLC}	地址 C 有效直到 ALE 变为低电平	22.5		$0.5t_{CLCL}$ −40.0		ns

表 15-2：时序参数表（续）

			8MHz 振荡器		可变振荡器		
标号	符号	参数	最小值	最大值	最小值	最大值	单位
5	t_{AVRL}	地址有效直到 RD 为低电平	95.0		$1.0t_{CLCL} - 30.0$		ns
6	t_{AVWL}	地址有效直到 WR 为低电平	157.5		$1.5t_{CLCL} - 30.0$		ns
7	t_{LLWL}	ALE 为低电平直到 WR 为低电平	105.0	145.0	$1.0t_{CLCL} - 20.0$	$1.0t_{CLCL}+20.0$	ns
8	t_{LLRL}	ALE 为低电平直到 RD 为低电平	42.5	82.5	$0.5t_{CLCL} - 20.0$	$0.5t_{CLCL}+20.0$	ns
9	t_{DVRH}	Data 设置直到 RD 为高电平	60.0		60.0		ns
10	t_{RLDV}	Read 为低电平直到 data 有效		70.0		$1.0t_{CLCL} - 55.0$	ns
11	t_{RHDX}	RD 为高电平后 Data 保持	0.0		0.0		ns
12	t_{RLRH}	PD 脉冲宽度	105.0		$1.0t_{CLCL} - 20.0$		ns
13	t_{DVWL}	Data 设置直到 WR 为低电平	27.5		$0.5t_{CLCL} - 35.0$		ns
14	t_{WHDX}	WR 为高电平后 Data 保持	0.0		0.0		ns
15	t_{DVWH}	Data 有效直到 WR 为高电平	95.0		$1.0t_{CLCL} - 30.0$		ns
16	t_{WLWH}	WR 脉冲宽度	42.5		$0.5t_{CLCL} - 20.0$		ns

由于处理器的所有操作都以时钟为基准，所以在图15-13和图15-14所示的波形图的上方都有系统时钟（system clock）。在Atmel的参考手册中，时钟周期用t_{CLCL}标识（注1），

注1：　技术手册中术语的命名方式常常模糊不清。术语CL是指Clock，由于Atmel使用4个下标字符来描述时序参考点，因此他们就用CLCL来表示Clock。你不用去了解这些下标的实际意思，只需知道这些下标所表示的信号以及相关的编码即可。

其值为频率的倒数。对一个 8MHz 的时钟来说，时钟周期等于 125ns。T1、T2 和 T3 的时钟宽度都为 t_{CLCL}。

处理器的时钟周期不会孤立地存在，它总是（注 2）有一个前导周期和一个后继周期。我们能从时序图中看到这一点。在新的访问周期开始时，上一次访问的地址仍驻留在地址总线中。在 T1 周期的下降沿，地址总线上的地址才变成本次访问的有效地址。端口 A 表示地址位 A0~A7，而端口 B 表示地址位 A8~A15。同时，**ALE** 变为高电平，释放外部地址锁存器，以便获得来自端口 A 的地址。**ALE** 维持高电平的时间为 $0.5t_{CLCL}-30ns$。举个例子，如果一个 AT90S8515 的工作频率为 8MHz，那么 **ALE** 保持高电平的时间为 32.5ns（0.5*125ns −30ns=32.5ns）。**ALE** 变为低电平时，外部锁存器便会得到并保持新的低 8 位地址。在 ALE 下降之前，地址位保持有效的时间为 $0.5t_{CLCL}-40ns$，换句话说，就是要在 T1 周期结束时，而且系统时钟上升之前保持 40ns。当 ALE 下降时，而且在端口 A 变为数据位之前，对于写周期，低地址位需要在端口 A 保持 $0.5t_{CLCL}+5ns$，但对于读周期，则只需要保持 15ns。在读周期时保持的时间要短得多的原因是，处理器希望只要有可能就释放信号引脚。因为读周期需要外部设备作出响应，这就意味着处理器必须尽可能地为响应信号释放出通道。

对一个写周期来说，在 **ALE** 下降后的 $t_{CLCL}-20ns$，写选通信号 $\overline{\textbf{WR}}$ 变为低电平。这对于外设来说，表明处理器已把有效的数据输出到数据总线上。$\overline{\textbf{WR}}$ 将保持低电平 $0.5t_{CLCL}-20ns$，这段时间允许外设做好将数据读入锁存器的准备。在 $\overline{\textbf{WR}}$ 的上升沿，外设被允许将数据总线上的数据锁存。此时，写周期结束，下一周期将开始。

对于一个读周期来说，在 **ALE** 变为低电平后的 $0.5t_{CLCL}-20ns$，读选通信号 $\overline{\textbf{RD}}$ 变为低电平。$\overline{\textbf{RD}}$ 低电平的持续时间为 $t_{CLCL}-20ns$，在这段时间内，外设应当将有效的数据放到数据总线上。只要能保证在 $\overline{\textbf{RD}}$ 再次变为高电平之前，数据能够稳定地保持至少 60ns，那么外设便可以在 $\overline{\textbf{RD}}$ 下降后的任意时刻发送数据。此时，处理器将锁存来自外设的数据，读周期终止。需要指出的是，很多处理器可能没有一个单独的读使能信号，因而它必须由外部逻辑电路产生。这基于一个前提，即如果一个周期不是写周期，那么它一定是读周期。

上述就是AT90S8515处理器如何访问其总线上连接的设备，不论这些设备是存储器芯片还是外围设备。但在实际应用中它又是如何工作的呢？下面我们将要看到的就是带有一些其他设备的基于 AT90S8515 的计算机的设计（注 3）。在这个例子中，我们将把处理器连接到一个静态 RAM 和一些用以驱动几组 LED 的简单锁存器上。

注 2： 在此，我忽略了复位或断电的前一刻所产生的情形。

注 3： 由于我们先前已经介绍过振荡器和电路编程，在此就不再赘述。但这并不意味着你应该从设计中丢掉它们。

内存映射和地址解码

对于处理器而言，它的地址空间是一个很大的线性区域。虽然处理器的地址空间也许由很多内部和外部设备组成，但在地址空间中它们并没有什么差别。对处理器来说，同样都是简单地执行存储访问操作。系统设计者（也就是你）的工作就是给每个设备分配存储区域，并提供地址译码逻辑电路。在对外访问期间，处理器将地址传送给地址解码器，由它来唯一地选择正确的设备（如图 15-15 所示）。如果我们的 RAM 占用了存储空间的一段地址区域，那么只要处理器访问该地址区域，RAM 就会被选中（而不是其他什么设备）。换句话说，当访问的地址不在这一区域时，RAM 就不应当被选中。

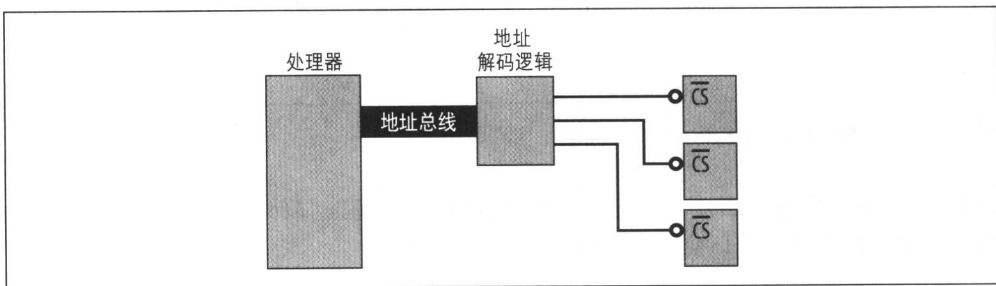


图 15-15：在地址空间中用于设备选择的地址解码器

在设备空间中对设备进行地址分配的过程就是所谓的内存映射或是地址映射过程。图 15-16 所示的是 AT90S8515 处理器的地址空间。我们所连接到处理器上的任何设备都必须映射到数据存储空间中，这样就可以忽略处理器内部的程序存储器。由于处理器采用的是哈佛结构，因而程序空间是一个完全独立的地址空间。有 64K 的数据空间用于处理器的内部资源：工作寄存器、I/O 寄存器以及 512 字节的内部 SRAM。它们占用了数据空间中的最低地址部分。任何大于 `0x0260` 的地址都可以由我们自由分配使用（注意：并非所有的处理器都采用这种内存映射方式，有些处理器的外围设备可以使用整个存储空间）。

现在，我们首要的任务是把剩下的地址空间分配给外部设备。由于 RAM 的大小为 32K，所以可以将它放在地址空间的高位地址部分（`0x8000~0xFFFF`）。如果设备地址处于整数边界，那么地址解码就会容易得多。把高位地址 `0x8000~0xFFFF` 分给 RAM，则低位地址部分用来存放锁存器和处理器的内部资源。一个锁存器只需要一个字节的地址空间。因此，如果有三个锁存器，那么我们只需给它们分配三个字节的地址空间即可。这就是所谓的显式地址解码。然而，可以通过一个很典型的例子来说明这种地址分配方式并不总是很有效的。假如，要对三个字节进行解码，那么将要求使用 14 位的地址解码器。但使用这么多的逻辑电路去选择三个设备显然没有必要。一个更好的解决方法是，将剩余的地址空间划分为四部分，其中的三部分分别用作锁存器，另一部分暂不使用

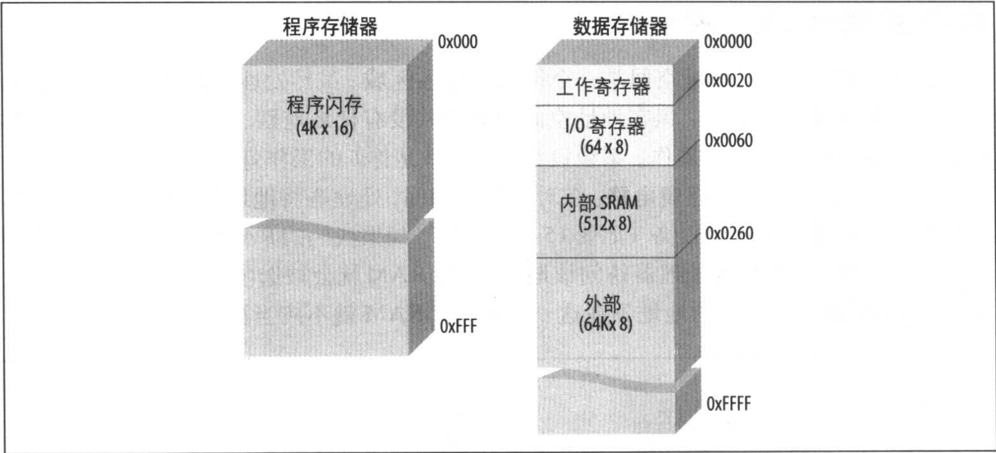


图 15-16：Atmel AT90S8515 内存映射

(留给处理器的内部资源)。这种方法被称为部分地址解码，显然它的效率更高。它的技巧就是使用最少量的地址信息来为你所需的设备解码。

图 15-17 中所示的是 SRAM 和三个锁存器的地址映射。我们注意到，最低地址部分 `0x0260~0x1FFF` 没有被使用。

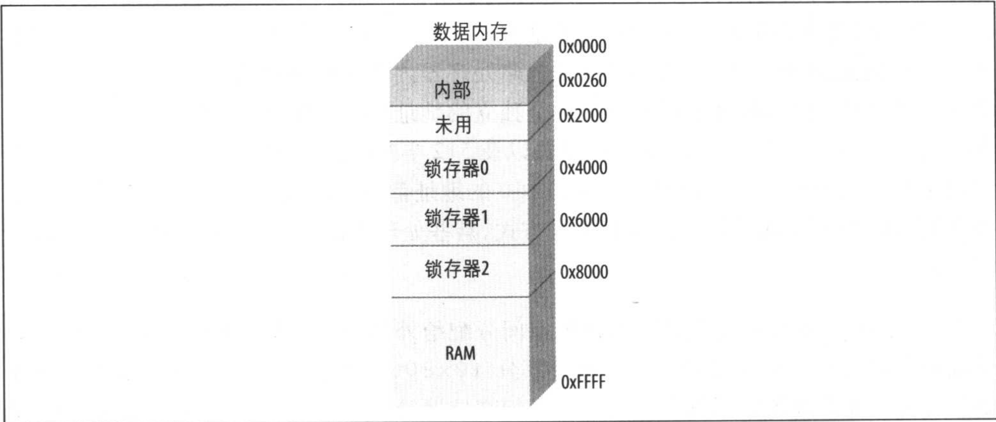


图 15-17：已分配的内存映射

任何 `0x2000~0x3FFF` 区域内的地址都将选择锁存器 0，即使是锁存器只需要一个字节的空间。因而，这个设备便被镜像到那个地址空间。为了便于编程，通常会选择地址(如 `0x2000`)，并且在代码中使用它。但同样也可以使用地址 `0x290F`，这并不会影响你的工作。

现在我们有了自己的内存映射，接下来需要设计的是一个地址解码器。我们现在列出设备以及它们相应的地址（见表 15-3）。我们需要查找的是：对于不同的设备，它们地址位中哪些是不同的；对于一个给定的设备区域，它们的地址位中又有哪些是相同的。

表 15-3：地址列表

设备	地址范围	A0…A15
未用	0x0000~0x1FFF	0000 0000 0000 0000~0001 1111 1111 1111
锁存器 0	0x2000~0x3FFF	0000 0000 0000 0000~0011 1111 1111 1111
锁存器 1	0x4000~0x5FFF	0000 0000 0000 0000~0101 1111 1111 1111
锁存器 2	0x6000~0x7FFF	0000 0000 0000 0000~0111 1111 1111 1111
RAM	0x8000~0xFFFF	1000 0000 0000 0000~1111 1111 1111 1111

那么对每个设备来说，什么是地址唯一性的关键呢？通过上表可以发现，对RAM来说，地址位（以及地址信号）**A15**是高电平，而对其他设备来说**A15**是低电平，因而可以使用**A15**作为选择RAM的关键。而对于锁存器而言，地址位**A15**、**A14**和**A13**是决定性的。于是我们可以重新绘制上述表格，使它的表述更为清晰。这也是更常用的构造地址列表的方法，见表 15-4。表中“x”代表无关项（即置 0 或置 1 均可）。

表 15-4：简化的地址列表

设备	地址范围	A0…A15
未用	0x0000~0x1FFF	000x xxxx xxxx xxxx xxxx
锁存器 0	0x2000~0x3FFF	001x xxxx xxxx xxxx xxxx
锁存器 1	0x4000~0x5FFF	010x xxxx xxxx xxxx xxxx
锁存器 2	0x6000~0x7FFF	011x xxxx xxxx xxxx xxxx
RAM	0x8000~0xFFFF	1xxx xxxx xxxx xxxx xxxx

因此，要解码RAM地址，我们只需要使用**A15**。如果**A15**为高电平，则RAM被选中。如果**A15**为低电平，那么其余的设备中将有一个被选中（当然RAM不会再被选中）。现在，RAM有一个片选信号$\overline{\textbf{CS}}$，是低电平触发。因而当**A15**位高电平时，$\overline{\textbf{CS}}$应该为低电平。所以，RAM的地址译码器只是简单地将**A15**反转，这可以通过使用一个反转芯片如74HCT04（如图 15-18 所示）来完成。通常在设备被选中后，片选信号都被标识，所以RAM的片选信号被标识为$\overline{\textbf{RAM}}$。

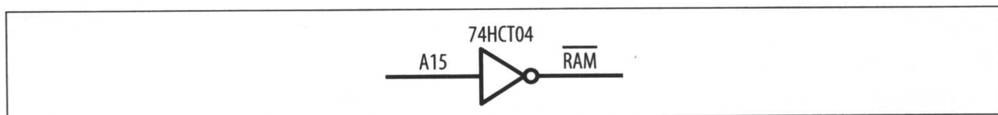


图 15-18：RAM 的地址解码器

需要指出的是，为了使能够对选择作出响应，RAM需要有一个片选以及一个来自处理器的读或写选通器。处理器的所有其他的地址线被直接连接到RAM的相应地址输入端（如图 15-19 所示）。

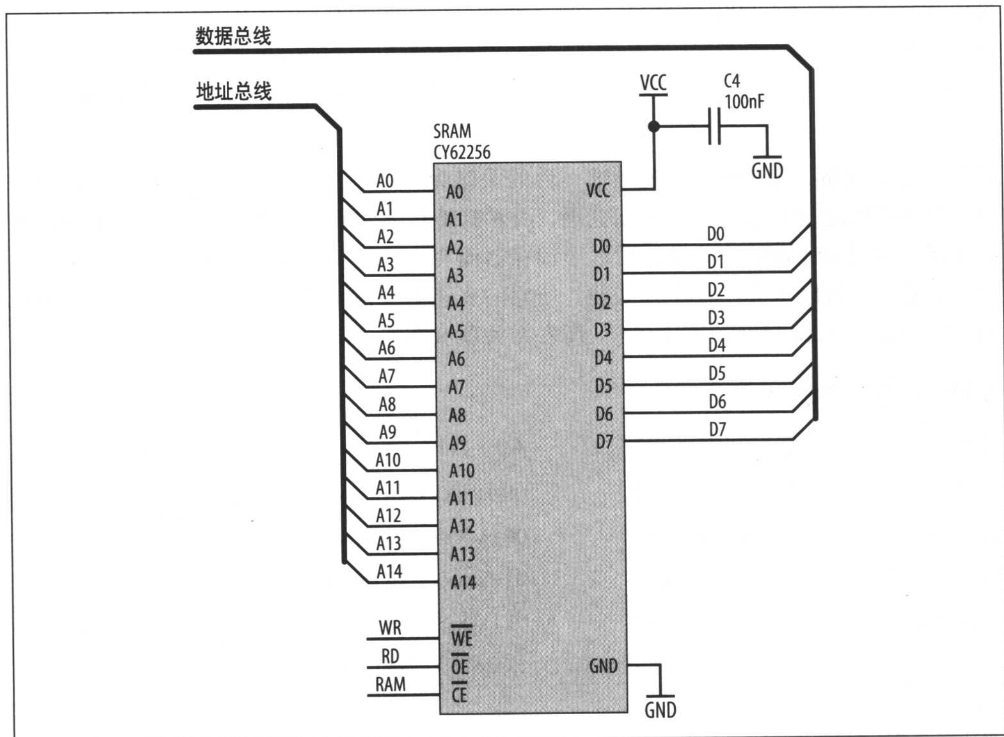


图 15-19：处理器与 SRAM 的连接

现在来看一下其余的 4 个区域。由于 **A15** 必须为低电平，所以 **A14** 和 **A13** 足够用来区分那些设备。如果我们的解码器使用离散逻辑电路，那它还需要一些门电路的支持，这可能会使电路变得很混乱。这里还有一个更简单的方法。我们可以使用 74HCT139 解码器（注 4），它有两个地址输入（A 和 B），能够产生四个唯一的低电平触发的片选信号输出（标记为 Y0~Y3）。因此，图 15-20 是计算机中所使用的完整地址解码器的电路图。

注 4：　在每一个 74HCT139 芯片中实际有两个独立的解码器，我们只用到其中的一个。

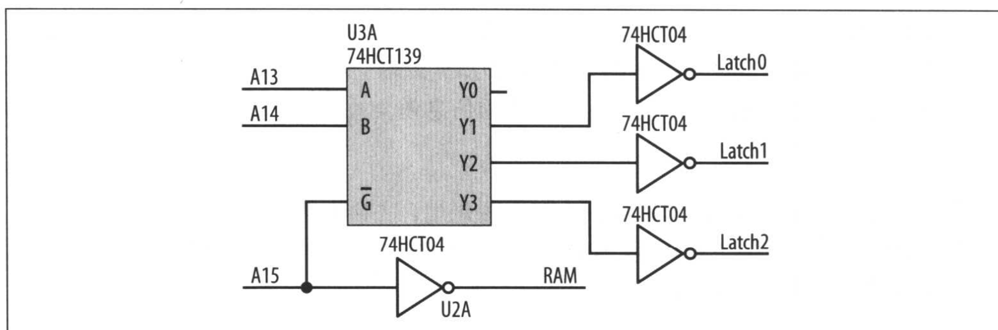


图 15-20：完整的地址解码器

74HCT139 解码器使用 **A15**（低电平）作为触发信号（输入端 $\overline{G}$），因此，**A15** 也被用来作为地址解码的一部分。如果我们要解码 8 个而非 4 个地址区，那么我们需要使用 74HCT138 解码器，它有 3 个地址输入端，可以产生 8 个片选信号。处理器与输出锁存器之间的接口很简单。我们可以使用同种类型的锁存器 74HCT573，它的作用是实现对地址的多路复用。这种输出锁存器可以应用于任何需要额外数字输出的环境中。在如图 15-21 所示的电路中，我使用 74HCT573 锁存器去控制 8 个一组的 LED。

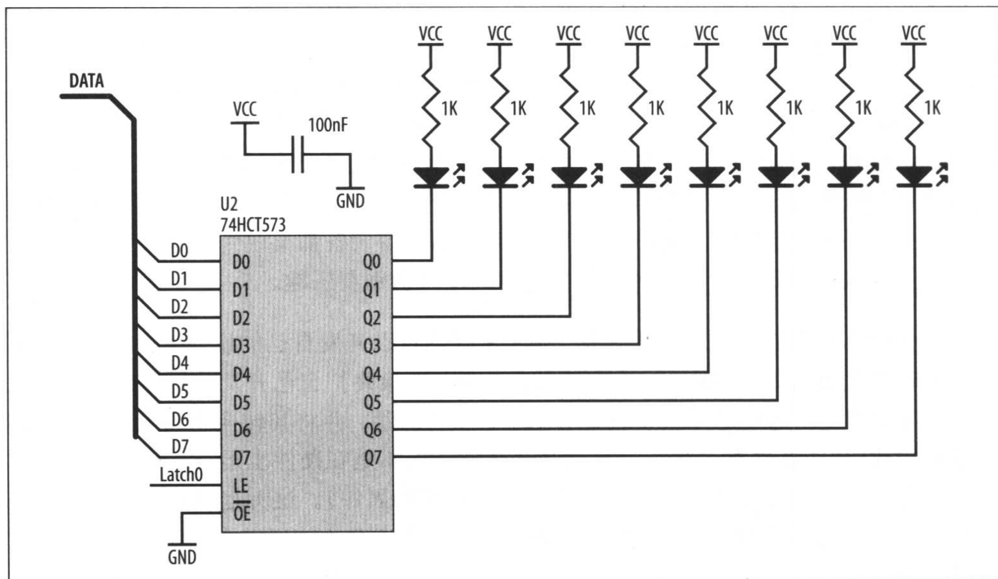


图 15-21：使用 74HCT573 锁存器控制 LED 组

来自 74HCT139 地址解码器的输出被用来驱动 74HCT573 锁存器的使能输入端 LE

(Latch Enable)。只要处理器访问那些分配给这种设备的存储区域，那么地址解码器将会触发锁存器以获取数据总线上的信息。因此，处理器会把一个字节的信息写入到锁存器地址空间中的任意地址上，接着这一字节的信息便会被输出到LED组（对应比特位为0的LED亮；为1则LED灭）。

需要指出的是，锁存器的输出使能端（$\overline{OE}$）恒接地。这意味着锁存器总是显示最后被写入的字节信息。这样做很重要，因为我们希望LED在处理器对它们访问的整个期间一直处于显示状态，而不仅仅是闪一下。

使用锁存器74HCT139的优点在于它可以简化我们的设计（和使用离散的逻辑门相比较而言），但仍有更好的方法可以实现系统的集成。

可编程逻辑PAL

现在很少看到由单独的门电路来支持逻辑实现。更常见的方法是使用可编程逻辑(PAL、LCA或PLD)（注5）来实现系统所要求的各种各样的混合功能。这种设备的特点是快速、紧凑、功耗低。而且由于它们可以重新编程，因而使系统设计更加容易，功能更多。

有很多种同类设备，从简单的芯片（可以用来实现胶合层逻辑电路，也是我们将要介绍的）到带有成百上千个门电路的大规模设备，都可供我们选用。Altera（*http://www.altera.com*）、Xilinx（*http://www.xilinx.com*）、Lattice 半导体公司（*http://www.latticesemi.com*）以及 Atmel 都是大规模可编程逻辑设备的开发商，生产这些大规模芯片的技术已相当成熟，以至于你甚至都可以将整个计算机系统集成到一个芯片中。软件核心是用门电路实现的处理器设计，是可以集成到这些可编程逻辑设备中的。你还可以将串行接口、磁盘控制器、网络接口以及一系列的外围设备都集成到这些大规模的设备中。当然，从底层开始设计和试验你自己的处理器也别有乐趣。

每一系列的芯片都有自己的一套开发工具，所以你可以开发自己的设计（使用电路图或使用某种编程语言如VHDL）来进行系统的仿真，并最终把自己开发的程序装载到芯片上。你甚至还能在这些芯片上装载C编译器来执行算法，并将其转换为门电路逻辑（而不是机器代码）。在这种情况下，算法是在软件，而不是在硬件上运行，但效果是一样的（即它已经成为具有软件功能的硬件，就是所谓的固件）。这听起来很酷，但是用以实现这种功能的工具却很昂贵。即使你要实现的只是一个小的嵌入式系统，也很可能会超出你的预算。而且对于我们的胶合层逻辑电路而言，这些芯片显然有点大材小用。由于我们需要的逻辑电路并不复杂，所以一个简单（而且廉价）的PAL就足够了，它还可以使用公共的、免费的软件进行程序设计。

注5：　分别为 Programmable Array Logic、Logic Cell Array 和 Programmable Logic Devices。

PAL 被设计使用等式来表示内部逻辑："+" 表示 OR，"*" 表示 AND，"/" 代表 NOT（这些运算符最初用于布尔逻辑中，如果你有程序设计的背景，那么这些符号可能有些古怪，因为你可能更习惯于使用"|"、"&"和"!"）。这些等式使用诸如PALASM、ABEL或CUPL等软件编译，生成一个JED文件。它使用了一个称为PAL的烧录器去配置PAL。在很多情况下，标准 EPROM 烧录器也用来给 PAL 编程。

PAL 有输出引脚和输入引脚，还有一些引脚既可用作输出，也可用作输入。你可以使用 PAL 的大部分引脚。在你的 PAL 源代码文件（PDS 文件）中，可以声明正使用的引脚，并标识它们。这与你在程序的源代码中声明变量很相似，唯一不同之处在于你不是给变量分配 RAM 上的字节单元，而是给芯片分配物理引脚。然后，你可以在等式中使用引脚标识去定义内部逻辑。我们的地址解码器（在 PAL 内部实现）将使用如下等式定义解码逻辑：

```
RAM = /A15
LATCH0 = /(/A15 * /A14 * A13)
LATCH1 = /(/A15 * A14 * /A13)
LATCH2 = /(/A15 * A14 * A13)
```

上述等式的表达方式可以使你很容易地将其和前面列出的地址列表相比较。你完全可以简化这些等式，但没有这个必要。正如一个优化的 C 编译器将会简化（并且加速）你的程序代码那样，PALASM 也会重构你的等式，使其更适于 PAL。

在 22V10 PAL 上编程实现前面所述的地址解码功能的 PDS 文件如下：

```
TITLE decoder.pds          ; name of this file  文件名
PATTERN
REVISION 1.0
AUTHOR John Catsoulis
COMPANY Embedded Pty Ltd
DATE June 2002

CHIP decoder PAL22V10      ; 定义使用的 PAL 设备种类
                           ;  并将其命名为("decoder")

PIN   2   A15              ; 引脚声明及分配
PIN   3   A14
PIN   12  LATCH0
PIN   13  LATCH1
PIN   14  LATCH2
PIN   15  RAM
EQUATIONS                  ; 等式开始处
RAM=/A15
LATH0=/(/A15*/A14*A13)
LATH1=/(/A15*/A14*A13)
LATH2=/(/A15*/A14*A13)
```

使用PAL设计系统逻辑电路有两方面的优点。当遇到bug或设计有变化时，PAL等式可以作出相应的改变。PAL的传输延迟相对固定而且短暂（无论是何种等式），这使得分析系统的时序信息变得简单多了。对于非常简单的设计而言，使用PAL或独立的芯片没有什么差别。但对于更复杂的设计来说，可编程逻辑电路是你唯一的选择。如果有可能的话，请尽可能地使用可编程逻辑电路而不是数字逻辑电路，因为可编程逻辑电路将使一切都变得更加简单。

时序分析

现在我们已经完成了逻辑设计，但问题是：它确实能工作吗？下面来分析一下它们的时序信息，这是最让人乏味，却也是计算机设计过程中最重要的一个环节。

我们首先要看到的是处理器的信号（和时序），然后是胶合层逻辑电路的作用，最后再看它们是否符合接口设备的要求。我们以SRAM为例进行分析，对于其他的设备，也可以用同一种方式进行分析。图15-22所示的是SRAM读周期的时序图。在此，我选用的RAM是由Cypress半导体公司生产的CY62256-70（32K）SRAM。大部分32K的SRAM都遵守JEDEC标准，这使得它们的引脚和信号都是相互兼容的。所以，对一种SRAM适用的操作就对其他的SRAM也同样适用。不过我还是要再次强调：在使用前，应该（且始终）先查阅所用设备的技术手册。

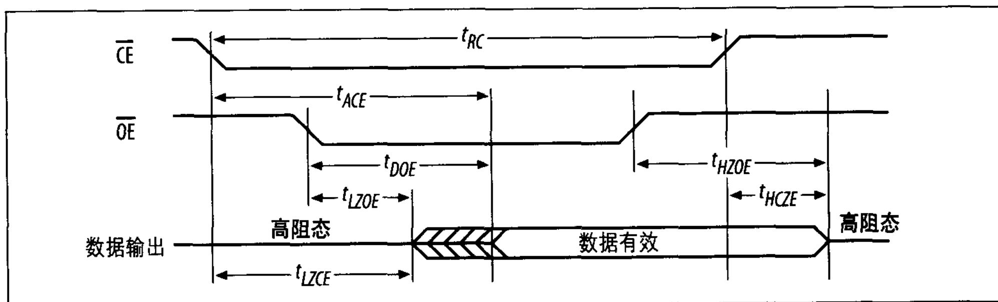


图15-22：RAM读周期的时序图

型号中“-70”表示这是一个70ns的SRAM，简单地说，就是对该芯片的访问时间为70ns。现在从CY62256-70的技术手册（可以从网站*http://www.cypress.com*获得）中，我们可以看到t_{RC}的最小值为70ns。这意味着芯片使$\overline{CE}$能可以保持低电平的时间要超过70ns。因为$\overline{CE}$正是来自地址解码器的片选（$\overline{RAM}$）信号，所以我们必须保证地址解码器能够使$\overline{RAM}$维持低电平至少70ns。要使SRAM在读周期内输出数据，必须有一个有效地址、一个激活的芯片使能信号以及一个激活的输出使能信号$\overline{OE}$。而输出使能信号$\overline{OE}$就是处理器的读选通信号$\overline{RD}$。上述三个条件都必须满足，芯片才可以输出数据。数据要在$\overline{CE}$

变为低电平（t_{ACE}）后的70ns或$\overline{\textbf{OE}}$被变为低电平（t_{DOE}）后的35ns（等两个条件都符合）才输出。现在，我们的信号$\overline{\textbf{CE}}$由地址译码器生成（反过来，地址译码器使用的又是来自处理器的信息），$\overline{\textbf{OE}}$（$\overline{\textbf{RD}}$）则来自处理器。在读周期的过程中处理器会输出读选通信号以及地址信息，以触发地址译码器。在读周期的后半段，处理器等待RAM将数据送到数据总线上。有一点非常关键：使RAM输出数据的信号必须保证处理器希望得到的数据必须是有效的。这就对RAM的数据输出触发信号提出了一个很严格的要求。如果能满足这个要求，那么你就可以得到一个能读取外部存储器数据的处理器；否则，你的处理器将毫无用处。

现在我们开始讨论处理器，我们假定它的等待状态产生器已失效。对于AT90S8515处理器来说，它的每个动作都是**ALE**的下降沿触发。在8MHz的AT90S8515芯片上，用以输入到地址解码器的高位地址将在**ALE**电平降低之前的22.5ns变为有效。如果我们使用的地址解码器对输入变化的响应时间为40ns（注6），那么RAM的片选信号会在**ALE**降低后的17.5ns变为有效，如图15-23所示。

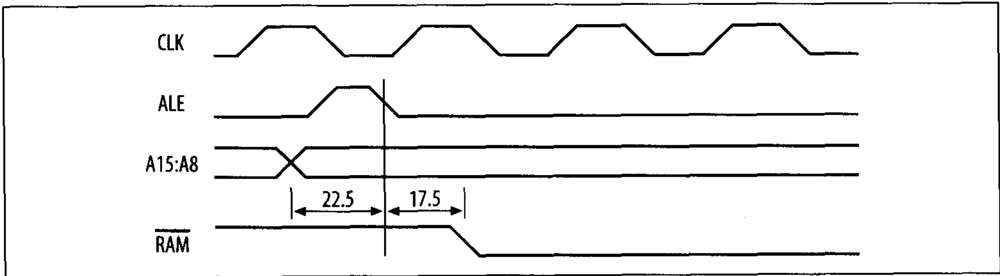


图15-23：RAM片选时序

而$\overline{\textbf{RD}}$将在ALE降低后的42.5ns~82.5ns内变为低电平。由于RAM直到$\overline{\textbf{RD}}$（$\overline{\textbf{OE}}$）为低电平才输出数据，所以我们在此取最坏的情况，即82.5ns，如图15-24所示。

由于RAM要在$\overline{\textbf{RAM}}$处于低电平状态70ns以及$\overline{\textbf{RD}}$处于低电平状态35ns之后才作出响应。我们看到，$\overline{\textbf{RAM}}$处于低电平70ns时**ALE**已变为低电平87.5ns，而$\overline{\textbf{RD}}$的低电平持续35ns时**ALE**已变为低电平117.5ns，所以在这种情况下$\overline{\textbf{RD}}$是起决定性作用的控制信号。这意味着SRAM将在**ALE**变低后的117.5ns输出有效数据，如图15-25所示。

注6：PAL可能在15ns或更少的时间内响应，这也就是PAL较离散逻辑成为更好选择的另外一个原因。

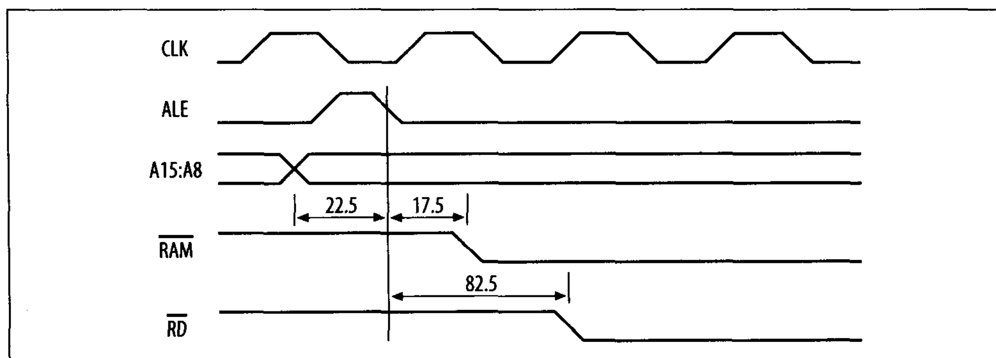


图 15-24：RAM 的读选通与片选

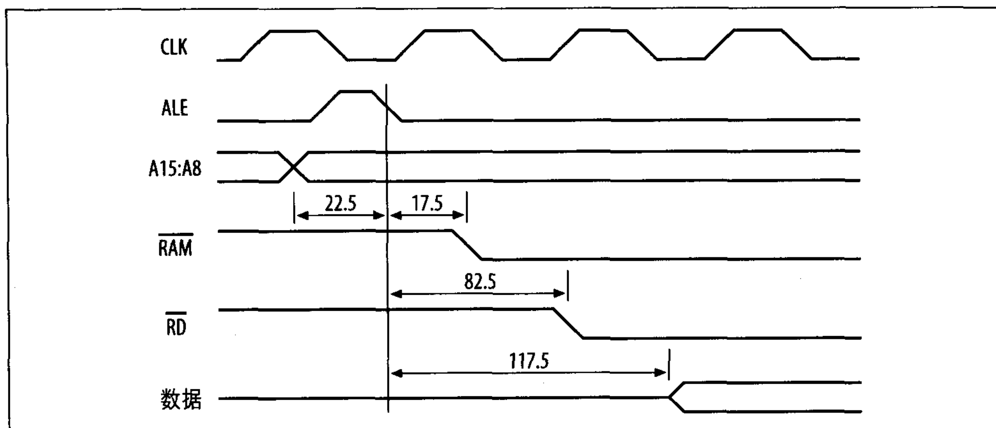


图 15-25：SRAM 的有效输出数据

现在，在 **ALE** 下降后 147.5ns 的读周期内，一个 8MHz 的处理器会将有效的数据锁存。因而，SRAM 将保留有效数据 30ns 以便处理器共享。到目前为止，一切设计都很完美。但在周期结束时又会怎样呢?这时，处理器希望数据总线能够被释放，以便下一次访问（即在 ALE 下降 200ns 之后）能够使用。RAM 被 **$\overline{\text{CS}}$** 释放直至它停止将数据发送到总线上，所需要的时间为 25ns。这意味着数据总线将在 142.5ns 时被 RAM 释放，这不会造成任何影响。

对写周期的分析也与上述过程类似。对每种不同的存储器访问周期以及对每一个连接到处理器上的设备作同样的时序分析非常重要。如果技术手册没有列出相关信息，或是虽有涉及但说得很不直接，这时分析时序是耗时和令人沮丧的，而且还很容易出错。然而我们却不得不做时序分析。否则我们只能寄希望于自己盲目的运气来使计算机运转，而且无论如何它都将不能算是一个好的设计。

存储器管理

在大部分的小规模嵌入式应用中，处理器与外部存储芯片都是直接连接的。然而，有时保持事物的原有次序是很有好处的，这就涉及到存储器的管理技术。

存储器管理要解决的问题是逻辑地址到物理地址的转换以及相反的过程。逻辑地址是处理器的输出地址，而物理地址则是存储器中将要被访问的真实地址。在小型计算机系统中，逻辑地址和物理地址往往是相同的，换句话说，系统不需要地址变换，如图 15-26 所示。

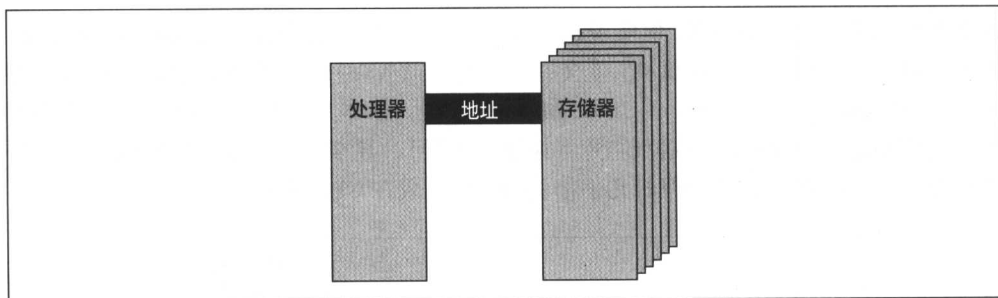


图 15-26：无地址变换方式

对小型计算机系统而言，没有存储器管理机制并不妨碍系统的运行。然而，在复杂的系统中，某种形式的存储器管理就可能很有必要了。以下四种情况中就需要存储器管理：

物理存储器 > 逻辑存储器

当处理器的逻辑地址空间（由地址总线的个数决定）小于系统的物理地址空间时，处理器的逻辑空间必须被映射到系统的物理存储空间之上。它有时被称为后备存储器（banked memory），听起来有些奇怪和陌生，但事实并非如此。我们常常会为了特定的目的而选择适用的处理器，不过这种处理器也许地址空间有限，对于你的应用而言也许太小了。通过后备存储器的使用，处理器的地址空间会扩大到超过逻辑地址范围的限制。

逻辑内存 > 物理内存

当处理器的逻辑地址空间非常巨大时，用物理存储空间去填充有时是不切实际的。磁盘上的一些空间也许被用作虚拟存储器，这样处理器看起来就拥有比芯片内部更多的存储空间。存储管理的任务是：判断存储访问的存储器是物理存储器还是虚拟存储器；将磁盘中虚拟存储空间上的内容转储到真正的内存中，并完成相应的地址变换。

内存保护

你也许想阻止一些程序访问某些存储区域。保护机制能够阻止那些恶意代码去破坏操作系统，使计算机崩溃。它也是使所有的I/O访问都通过操作系统来进行的一种方式，因为保护机制可以阻止软件（操作系统除外）对I/O空间的访问。

任务隔离

在多任务系统中，任务间彼此不应该受到干扰（如越界访问其他任务的内存空间），而且两个不同的任务应该能够使用相同的逻辑地址，可以通过内存管理机制将逻辑地址变换为各自的物理地址。

存储管理的基本思想很简单，但实现起来却很复杂。而且存储管理技术也很多，可以说，有多少种使用存储管理的计算机，就会有多少种存储管理技术。存储器管理由存储器管理单元（Memory Management Unit，MMU）实现，基本形式如图 15-27 所示。MMU 可以是商用芯片，也可以是定制芯片（或逻辑电路），甚至还可以是处理器内部的一个集成模块。而现在，高速处理器几乎都将 MMU 集成在 CPU 芯片上。

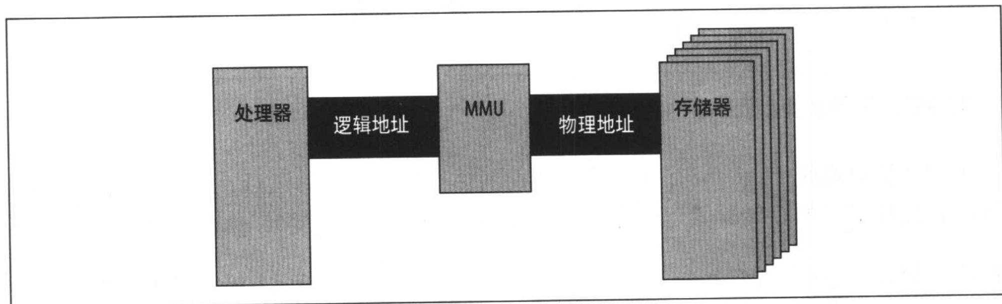


图 15-27：使用 MMU 的地址变换

页面映射

在所有实用的存储管理系统中，内存中的字按页分组，因而地址也可以看作是由两部分组成：页号和页内的字号。MMU 将逻辑页转换成物理页，而字号则无需改变（如图 15-28 所示）。实际上，整个地址就是页号与字号的拼接。

来自处理器的逻辑地址被分为两部分：页号和字号（页内偏移地址）。页号由MMU变换得到，再复合字号就可以形成存储器内的物理地址，如图 15-29 所示。

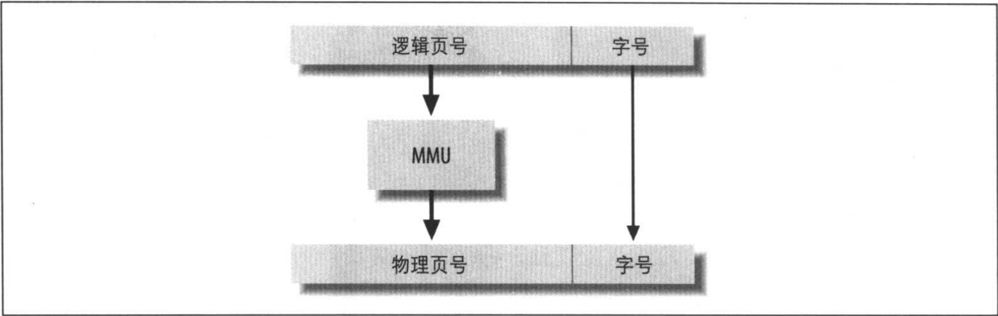


图 15-28：地址变换

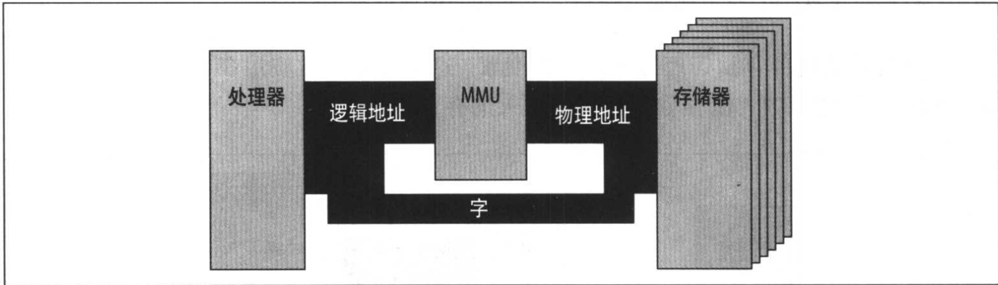


图 15-29：使用页地址转换的系统

后备存储器

当逻辑地址空间小于物理地址空间时，存储管理形式最为简单。如果系统将每一页的大小设计成等于逻辑地址空间，那么只要由 MMU 提供页号，就可以把逻辑地址映射为物理地址，如图 15-30 所示。

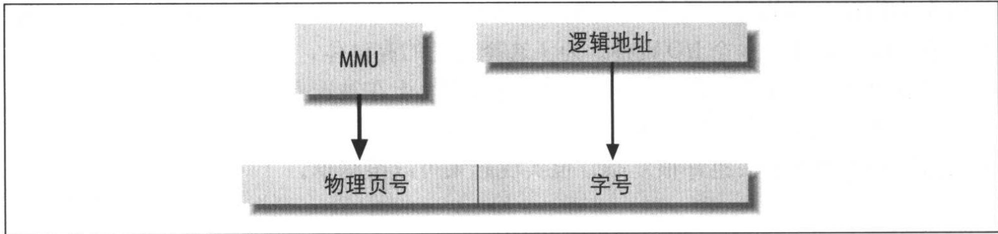


图 15-30：MMU 生成页号

图15-31所示是使用这种存储管理方式的有效地址空间。逻辑地址空间可以被映射到（以及重映射）到物理地址空间的任何地方。

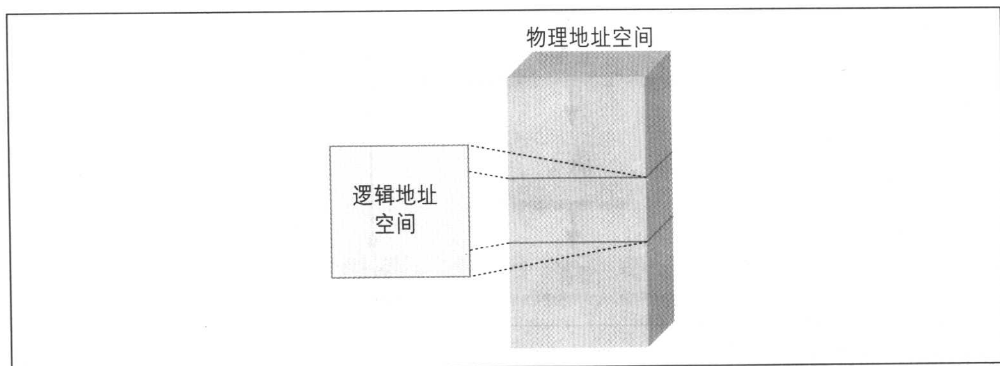


图 15-31：一个较小的逻辑地址空间到一个较大的物理地址空间的映射

图 15-32 所示是这种存储管理方式的系统配置。这种技术通常应用在 16 位地址（64K 逻辑空间）的处理器中，以便它们能访问更大的存储空间。

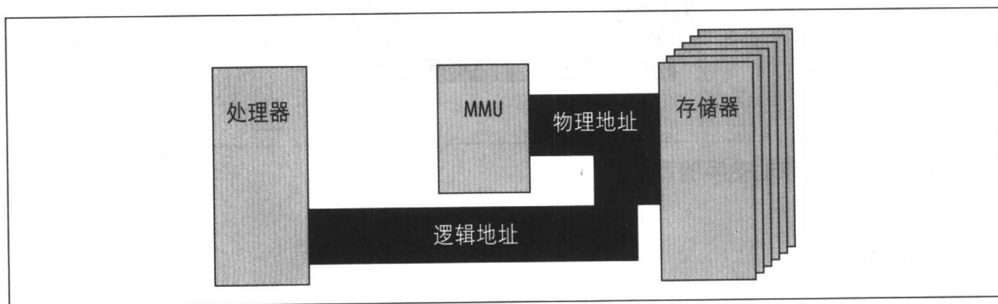


图 15-32：更大的物理地址空间的生成

对很多小型系统来说，后备存储器也可以通过锁存（请求并且占有）数据总线，将数据总线作为物理内存的附加地址来实现，如图 15-33 所示。对于处理器的逻辑空间来说，锁存器可以被看作另一个 I/O 设备。为了选择合适的存储体，处理器将存储体比特位放到锁存器中锁存。所有对逻辑地址空间的访问都发生在被选择的存储体内。在本例中，处理器的地址空间作为一个 64K 的窗口以映射到更大的 RAM 芯片上。如你看到的那样，虽然存储器管理看上去也许很复杂，但实现起来可以很简单。

注意： 这种技术也被应用于桌面系统。老式的苹果机带有一个 256K 的内存，不过它的 6502 处理器的逻辑地址空间却只有 64K 。

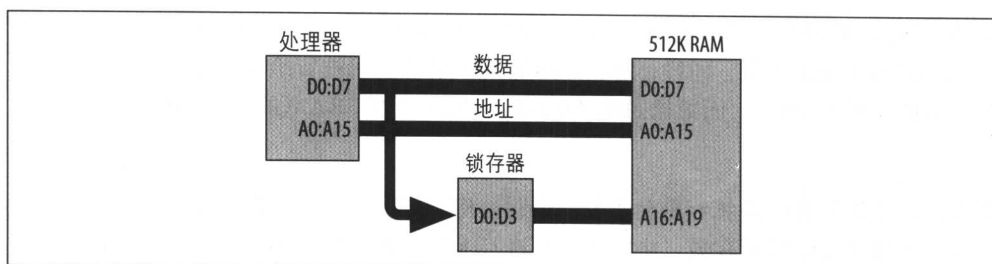


图 15-33：简单后备存储器的实现

图15-34所示是在AT90S515系统中实现后备存储器的布线要求，在本例中用一个512K的RAM代替了32K的RAM。

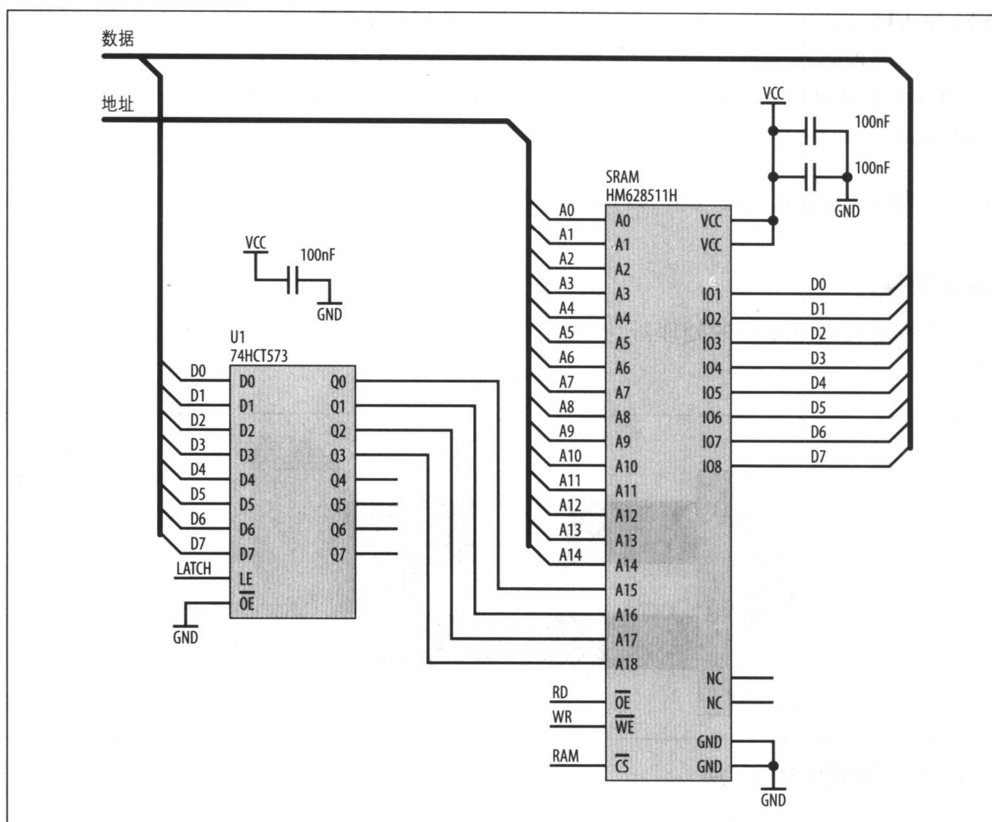


图 15-34：AVR 计算机的后备存储器

其中使用的RAM是由日立公司生产的HM628511H。在本例中，我们仍和以前一样将

RAM 放在处理器的高 32K 地址空间。换句话说，处理器的高 32K 地址空间可以映射到 RAM 的 512K 地址空间。同样，处理器的低 32K 地址空间被留给 I/O 设备。地址位 **A0**~**A14** 仍被连到 RAM 上，而数据总线（D0~D7）则被连到 SRAM 的数据引脚（**IO1~IO8**）（注 7）。

现在，如同在应用 LED 锁存器时那样，我们也将 74HCT573 锁存器映射到处理器的地址空间。处理器可以写锁存器，并且锁存器将写入的数据保持在输出端。锁存器的低半字节被用来给 RAM 提供高阶地址位。

来看一下处理器访问地址 `0x1C000` 时的情况。地址 `0x1C000` 写成二进制形式为 `%0001 1100 0000 0000 0000`。低 15 位地址（**A14~A0**）由处理器直接提供，而其余的地址位则必须被锁存。所以，处理器首先将字节 `0x03` 锁存，即锁存 RAM 的 **A18**、**A17**、**A16** 和 **A15**（见 `%0011`，即 `0x03`）地址引脚。RAM 的这个区域被用作处理器的 32KB 窗口。当处理器访问地址 `0xC000` 时，高阶地址位（**A15**）被地址译码器使用，以选择 RAM（使其片选信号 $\overline{\textbf{CS}}$ 输出低电平）。而其余的 15 位（**A0~A14**）再加上锁存器的输出即可选择地址 `0x1C000`。

NC 引脚的意思是无连接（No Connection），没有线路连接它。

地址变换

对于具有更大地址空间的处理器，MMU 可以提供地址总线上半部分的变换，如图 15-35 所示。

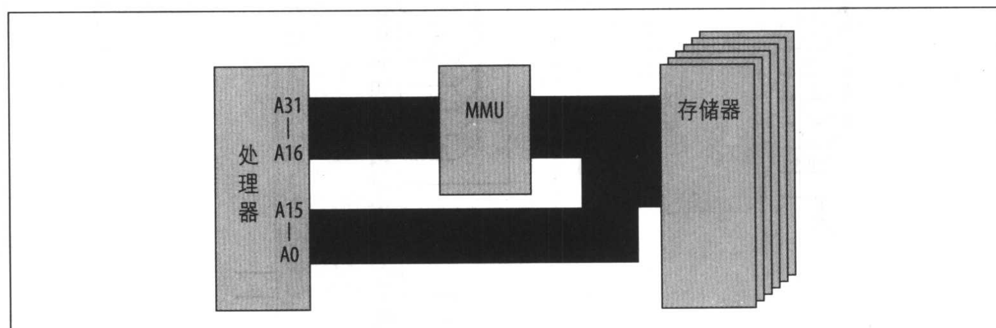


图 15-35：逻辑页号的变换

注 7：由于这些引脚为设备完成数据输入/输出工作，存储芯片制造商经常标注数据引脚为 I/O 引脚。

MMU 包含一个地址变换表，可以把输入地址重新映射到不同的输出地址上。为了改变地址变换表，处理器必须能够访问 MMU（如果地址变换表不能更改，那么 MMU 也就没有存在的必要了）。一些处理器可以与专门的外部MMU协同工作，而另一些处理器则把 MMU 集成在芯片内。但如果处理器在设计时没有考虑到要使用 MMU，那么它就没有对 MMU 的特定支持。这样一来，处理器与 MMU 的通信方式就和其他的外设通信没有区别，同样要使用标准的读 / 写周期。这也就要求 MMU 必须出现在处理器的地址空间。看起来最简单的方法是将 MMU 映射到系统的物理地址空间中，遗憾的是，在实际中这并不可行。因为一旦（有意或无意地）MMU 不能映射到当前的逻辑地址空间（原因可能是MMU所在的物理页面不在当前逻辑地址空间的地址范围内），那么MMU就不能够再被访问到。这也可能发生于系统上电时，因为这时 MMU 的地址转换表的内容可能是未知的。

解决的办法是 MMU 的片选信号由处理器的逻辑地址总线直接判定。因此，MMU 将位于逻辑地址空间的一个恒定地址区域。不过，这种方法虽然避免了MMU的“丢失”，却引入了一个新的问题。由于MMU 直接处于逻辑地址空间，那么不可避免的，它将受到突发性事件的影响（由系统崩溃所引起），或是在多任务的环境中受到一些非法的、恶意的干扰。为了解决这一问题，很多大的处理器都使用了两种状态的操作方式：管理态和用户态。这两种操作模式分别使用了不同的堆栈指针。这就在操作系统（以及它的特权程序）和系统运行的一般任务之间形成了屏障。MMU 通过处理器上专用的状态引脚获知处理器的状态。只有当处理器处于管理态时，MMU 才能够被修改，这就限制了用户程序对其进行修改。对不同的工作态，MMU 使用不同的逻辑 - 物理地址变换表。管理态变换表通常在系统初始化时被配置，之后就保持不变。用户任务（用户程序）通常运行在用户态，而操作系统（用来执行任务切换和处理 I/O）则运行于管理态。中断程序也会把处理器置于管理态，这就使得向量表和服务例程可以不在用户的逻辑地址空间。而在用户态，操作系统可以使某些特殊的物理页不被用户程序所访问。

第 16 章

68HC11

人无法两次踏入同一条河流，
因为无论是这条河还是这个人都已经不同。
——*赫拉克利特*
《On the Universe》

在这一章里，我们将要看到的是 Freescale Semiconductor（前身为 Motorola）公司的 68HC11，这个处理器的体系结构又让我想起了微控制器早期的时候。我对这个体系结构怀有眷恋之情。最早我就是在基于6802处理器的机器上学会编写汇编语言的，至今我还记得其中许多操作码，我可以把 6800 的机器码直接当作原始码来读。

虽然这不是什么新体系结构，但是它容易编程、容易构建，且已经被稳定地使用了很久。它是一个可以立即上手的好平台，使用它可以马上拼凑出一个 8 位计算机。现在让我们来快速的概览一下这个处理器的体系结构。

68HC11 的体系结构

MC68HC11 是 6800 系列 8 位处理器的成员之一。68HC11 是一个高密度的 HCMOS 微控制器单元（microcontroller unit，MCU），采用全静态设计技术。它基本上是一个标准的 6800 处理器（经过若干强化）再加上内部集成的外部设备，包括一个经过强化的 16 位定时器（具有 4 个可编程计数器）、一个串行接口（SPI）、一个串行通信接口（SCI）、一个 8 位的脉冲累加器、实时中断、板上静态 RAM、一个 8 通道 ADC 以及板上 EEROM。

图 16-1 所示为 MC68HC11 的主寄存器（注意，此处并未涵盖芯片内与各个外设有关的控制寄存器）。

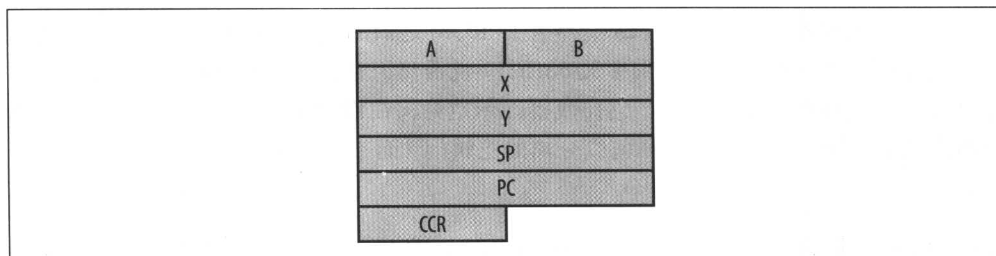


图 16-1：MC68HC11 的寄存器

MC68HC11 具有两个累加器：A 和 B。这两个累加器的宽度都是 8 位，不过我们可以把它们当成一个 16 位的累加器 D 来使用（如图 16-2 所示）。

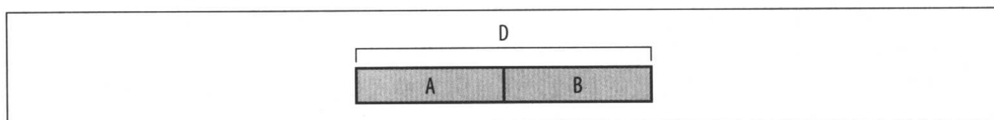


图 16-2：MC68HC11 的累加器

索引寄存器（X 和 Y）是 16 位的，可以用来存放地址。因此，它们可用来指向存储器中的某个位置。它们也可以被用作16位的计数器或临时存储寄存器。程序计数器（program counter，PC）是一个 16 位的计数器，它会指向下一个指令在存储器中的位置（换言之，它所存放的就是下一条指令的地址）。条件码寄存器（condition code register，CCR）是一个特殊的 8 位寄存器，它会显示处理器的当前工作状态。

堆栈指针（stack ponter，SP）是一个 16 位的寄存器，用来指向堆栈中下一个可用的位置。堆栈是一段用来存储数据或地址（视为数据）的存储器。当一个值被压入堆栈时，此值会被存放到堆栈指针所指向的位置。接着堆栈指针就会被自动递减，并指向下一个可用的位置。当有数据从堆栈中取出时，堆栈指针就会被自动递增，接着从该位置取出数据。当68HC11的堆栈被压入数据时，堆栈在存储器中就会往下（低地址方向）增长。当一个 16 位的值被压入堆栈时，堆栈的指针就会被递减两次（两个 8 位数据所占的空间）。

以上是68HC11的基本编程模式。尽管功能不是很强大，但是它简单又好用，而且很容易上手。现在让我们来了解如何构建一个基于 68HC11 的计算机。

一台基于 68HC11 的简单计算机

这台计算机会有 32KB 的静态 RAM、16K 的 EPROM、一个串行接口（68HC11 内部集

成）以及经锁存器控制的一组LED。尽管EPROM是旧的技术，基于两个理由，我还是位系统选择了此元件。首先，大多数68HC11机器仍旧在使用EPROM，通常是由于系统历史性和传统的因素。其次，它让我得以示范如何在设计中使用EPROM。除了告诉你如何设计之外，我还会介绍如何以分离的逻辑门（而不是PAL）来为计算机完成组合逻辑。

68HC11被设计成可以使用在各种小型的应用上，有许多都跟外部设备的监视和控制有关。就其本身而言，它可以执行在以下的模式：单芯片（single-chip）模式、扩展多工（expanded multiplexed）模式、自引导（bootstrap）模式以及测试（test）模式。最后一个模式只供Freescale在制造期间使用，而不推荐用户使用。在单芯片模式下，68HC11的操作完全靠内部的功能（小型的RAM、小型的ROM、I/O以及定时器），不会使用外部的地址或数据总线。因此，大部分引脚（也就是这个处理器的A、B、C和D端口）都只具备数字I/O的功能。在扩展多工模式中，68HC11的行为如同一个普通的8位处理器，以B和C端口作为地址和数据总线。在自引导模式中，68HC11会从内部的192字节的ROM中载入它的向量，并初始化它的内部串行接口。在软件控制下，68HC11可以从自引导模式切换到任何其他模式。68HC11上有两个特殊的引脚（**MODA**和**MODB**），可用来决定处理器将会进入何种模式。表16-1所示为**MODA**和**MODB**的设置及它们对68HC11所造成的影响。

表16-1：68HC11的开机模式

MODB	MODA	模式
1	0	单芯片
1	1	扩展多工
0	0	特殊自引导
0	1	特殊测试

因为我们想要添加外部的存储器和锁存器，所以68HC11必须进入扩展多工模式。因此，在我们的设计中，**MODA**和**MODB**都必须接高电平。

为减少68HC11外部引脚的数目，Freescale以多工的方式让相同的引脚提供地址和数据总线。这意味着芯片将会比较小（因此也比较便宜），不过需要系统设计者加入外部的逻辑，以便从多工总线复用（分离）出地址和数据总线。数据总线与地址总线的下半部分共享C端口，而地址总线的上半部则独立于B端口，所以不需要复用。一个特别的输出端**AS**（adress strobe，地址选通）用来指示目前数据总线上所出现的是地址信息还是数据信息。

图 16-3 所示为存储器周期的时序图。地址在 AS 变为高电平之后生效，并持续到 **AS** 的下降沿。**AS** 可作为锁存器的控制输入端，用来复用地址总线的下半部。一旦锁存，锁存器就会持续输出地址，直至整个周期。稍后数据就会出现在多工总线上。

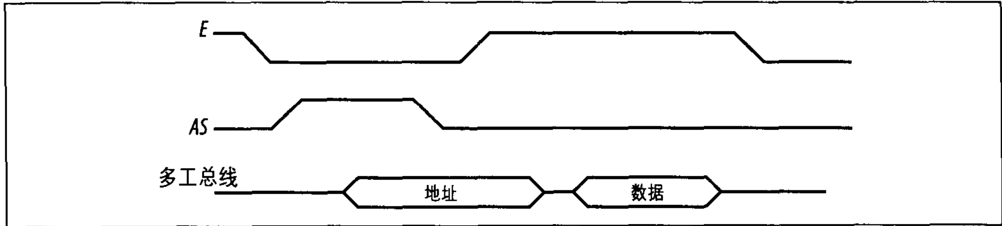


图 16-3：MC68HC11 的多工总线的时序

在 E 变为高电平（位于此周期中间）之前 94ns，上半部的地址位（**A8:15**）会出现在 B 端口，直至整个周期。这些地址线不需要复用。

图 16-4 所示为扩展多工模式下的 68HC11 处理器的基本电路，其中包括下半部地址线的复用。中断输入端 **IRQ** 和 **XIRQ** 都需要上拉电阻。Motorola 建议使用 MC34064 芯片来产生上电复位信号。这个只有三根引脚的简单元件仅需要电源和接地，而且将会在通电时产生一个可供 68HC11 使用的复位脉冲。为了给处理器提供时钟，我们要为处理器的内部振荡器（引脚 **XTAL** 和 **EXTAL**）添加 8MHz 的晶体。这个晶体需要两个旁通电容 C1 和 C2，以及一个并联电阻 R1。

D 端口可用作内部串行端口，第 0 位是接收数据的输入端（$\mathbf{R_X}$），第 1 位是传送数据的输出端（$\mathbf{T_X}$）。D 端口还包含了处理器的 SPI 接口，这让处理器可以轻而易举地连接各种外部设备。

这就完成了处理器的基本需求。下一个工作就是设计计算机的其余部分，这对我们的系统而言非常简单，就是一个 RAM、一个 ROM 以及一个锁存器。首先我们需要配置存储空间，然后再设计地址解码器。

地址解码

图 16-5 所示为 MC6800 和 MC68HC11 的地址空间。它们各自都具有 64K 的存储空间（16 位地址），不过 68HC11 在它的存储空间配置图中还额外多了若干内部功能。寄存器块不是由我们前面提到的累加器或索引寄存器所构成，这些元件并不会出现在存储空间配置图中。寄存器块是一些特殊的寄存器所构成的阵列，专门用于控制处理器所具有的许多内部设备，如计数器 / 定时器、AD 转换器等等。请注意，68HC11 能够通过软件把 I/O 寄存器以及 256B 的 RAM 重定位于任意 4K 的范围之内。这意味着设计者可以针对自己

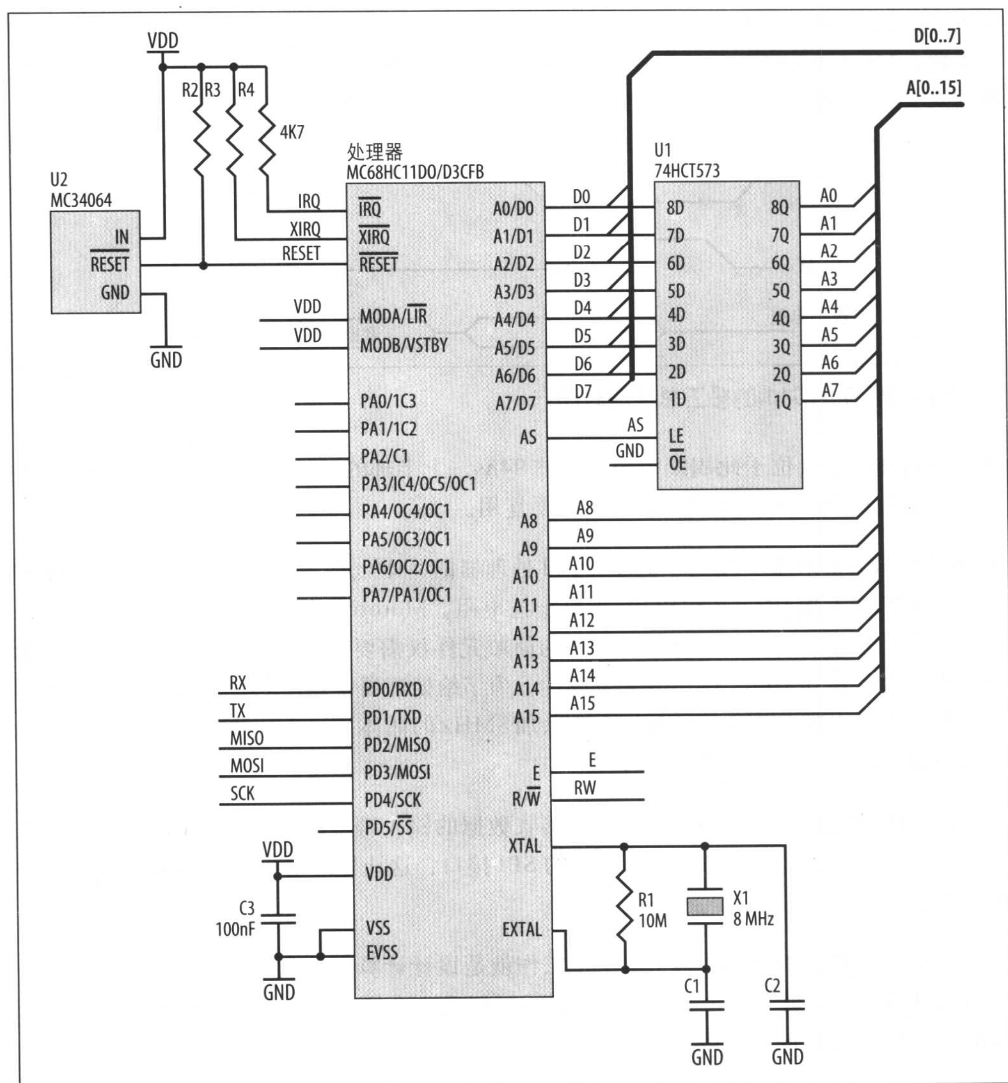


图 16-4：MC68HC11 与支持元件

的设计，将这些元件摆在最适当的地方。图 16-5 中标识为 External 之处表示此范围内的地址可供外部存储器或其他外设所使用。

向量是一个地址表，用来指向存储器中的例程。其中最重要的是位于地址 `0xFFFE` 的 16 位 RESET 向量。此位置包含了一个 16 位的指针，指向存储器中初始化程序代码所在的位置。处理器会在上电或复位之后把这个指针载入到它的程序计数器中，从而开始软

图 16-5：比较 6800 和 68HC11 的存储空间配置

件的执行。因为系统上电的时候需要此向量有效，所以它必须是非易失的（即使断电也不丢失）。基于这个原因，通常会有一个 ROM 被放在地址空间的最上层区域。

因为 68HC11 具有 64K 的地址空间，所以 32K 的 RAM 将会占用一半的地址空间。不过是哪一半呢？正如前文所述，未供电期间向量表会被保存下来，所以必须有一个 ROM 被放在地址空间的最上部。因此，我们的 32K RAM 必须放在地址空间的下半部分。然而，内部的RAM以及外设的寄存器也会被映射到地址空间的下半部。如果我们将这32K 的 RAM 映射在此空间，会不会引起冲突？（换言之，我们是否需要特殊的逻辑解决此问题？）答案是否定的。内部RAM和I/O寄存器具有优先权，而且存取它们所在的位置也不会在处理器外部总线上引起任何活动。实际上，它们与外部RAM的顶端彼此重叠。从我们的角度来看，这可以简化我们的设计，因为不需要使用特殊的逻辑来从我们的存储空间中排除这些地址。并且由于内部外设寄存器可以在软件的控制下重新映射，它们将被转移到地址空间中未被使用的其他地方。

本例中32K RAM具有的地址范围为`0x0000~0x7FFF`。在二进制中，就是`0000 0000 0000 0000`到`0111 1111 1111 1111`。此范围内的任何地址比特的组合都会被映射到RAM。与地址线 A0~A14 相关的存储器地址就在这个 RAM 里面，因此不必使用地址解码器。换言之，地址解码器并“不在乎”它们的状态，所以会被直接连接到 RAM。表 16-2 为这个 RAM 的地址表。其中，X 代表“不在乎”。

表 16-2：RAM 的地址位使用情况

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RAM	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

看一下该RAM的地址范围：`0000 0000 0000 0000~0111 1111 1111 1111`。所有的地址位都有一个共同的特点：A15 位为 0。因为这个 RAM 的大小为 32K，处理器的64K 地址空间被占用了一半，所以这个 RAM 的解码只需考虑 A15 地址位。因为这个 RAM 被放在地址空间的下半部，所以当最高地址位 A15 是低电平时，RAM 将必须被选中。当它为高电平时，这个 RAM 将不会被选中。这个 RAM 的解码因此可以被简化成由 A15 直接连接到 RAM 的 $\overline{\mathbf{CE}}$。还有比这更简单的么？图 16-6 所示为这个 RAM（以 62256 SRAM 为例）的电路。这个通用的电路可以用于任何标准的 32K × 8 RAM。

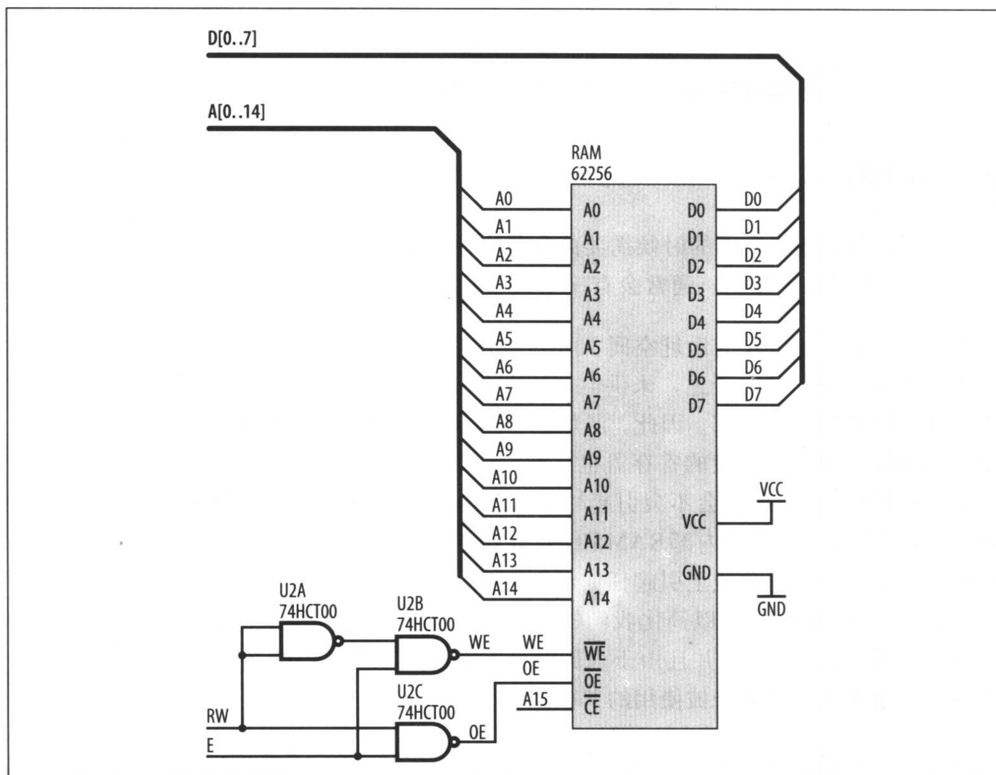


图 16-6：RAM 电路

当 A15 为低电平时，$\overline{\mathbf{CE}}$ 被置为低电平，所以 RAM 被选中（因为它是低电平使能引脚）。因此，当 A15 为低电平时，RAM 会被启用；当 A15 为高电平时，RAM 不会被启用。

注意电路中额外的逻辑（三个与非门）。68HC11 所产生的 $\mathbf{R/\overline{W}}$ 信号，可用来指示目前是读周期（$\mathbf{R/\overline{W}}$ 为高电平）或是写周期（$\mathbf{R/\overline{W}}$ 为低电平）。因为 RAM 对读和写的存取控制使用独立的输入端，而且它们都是低电平有效，所以此逻辑会将单一的 $\mathbf{R/\overline{W}}$ 信号转换成 RAM 的两个输入端：$\overline{\mathbf{WE}}$（write enable）和 $\overline{\mathbf{OE}}$（output enable）。$\mathbf{R/\overline{W}}$ 信号会与处理

器的E时钟组合在一起，以确保这两个输入端会在有效期间（E为高电平）起作用。否则，当不是写周期的时候，$\overline{\text{OE}}$也会起作用。与非门（NAND）UZA的功能只是一个反相器。

ROM是一个16K的器件，它是其余地址空间（64KB地址空间的上半部）的一半。最后只剩下锁存器这个唯一的其他外部器件了（它只需占用一个字节）。有两种做法可以将剩余的32K存储空间配置给这两个器件。第一种做法是采用“全地址解码”，也就是把每个地址位纳入考虑范围。采用全地址解码，锁存器将只会占用一字节的地址空间。所以，如果我们决定将锁存器放在地址0x8000，则每个地址位的状态如表16-3所示。

表16-3：选取0x8000上的锁存器的地址位

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RAM	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

如果采用全地址解码，我们必须在地址解码器中使用每个地址位。这样的逻辑不仅复杂，而且效率不高。

比较好的做法就是采用“部分地址解码”。此方法就是只使用足以区别这两个器件的地址位，来进行存储空间的分配。如果所分配的地址空间大于器件的需求也不会有什么问题。因此，即使分配16K的存储空间给锁存器，当我们选中锁存器时，它仍旧能正常工作。尽管这么做相当浪费地址空间，不过相对于“全地址解码”方式，这是个非常有效率的做法（就逻辑而言）。如果你使用的是离散的逻辑门，传播延迟也会因而降低，那么就不必考虑逻辑的问题。当为微处理器系统设计任何逻辑时，时序是不容忽视的问题。如果时序不对，系统就无法运作。

所以，剩下的32K地址空间需要分配给这两个器件。表16-4所示为RAM、锁存器和ROM等三个器件的地址表。

表16-4：所有器件的地址位分配

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RAM	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
锁存器	1	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X
ROM	1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	X

在此表中，锁存器和ROM都被分配到了16K的地址空间。换言之，将剩下的32K地址空间平均地分配给了这两个器件，这将使得地址的解码变得简单许多。图16-7所示为此计算机最终的存储空间配置图（此图忽略了内部I/O寄存器和存储器）。

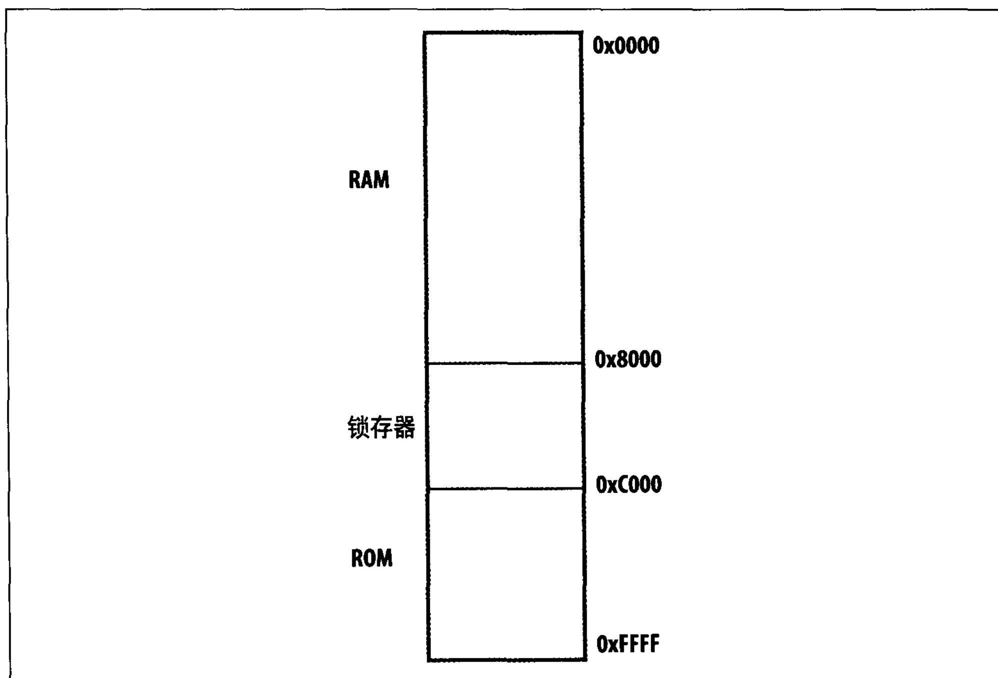


图 16-7：使用部分地址解码方式的地址空间分配

请注意，因为我们使用的是“部分地址解码”，锁存器将会多次出现在配置给它的空间中。锁存器其实是地址 0x8000 上的一个字节，不过它的地址解码只用到了**A15**和**A14**，所以任何地址只要**A14**为低电平、**A15**为高电平，都会选中它。因此，锁存器会出现在范围为 0x8000~0xBFFF 的地址空间内。举例来说，如果我们存取 0x9401 这个位置，该地址的**A14**为低电平、**A15**为高电平，就会选中锁存器。因为解码器并未使用A0~A13的地址线，所以它们的状态与地址的解码无关。

图 16-8 所示为LED锁存电路原理图（U3 本身的电源与接地其实是存在的，但此图并未明确给出）。此锁存器具有两个控制线 **LE** 和 **$\overline{\text{OE}}$**。只要 **LE** 从高电平变成低电平，就会使锁存器捕捉到并保存目前的输入数据，也就是处理器的数据总线。**$\overline{\text{OE}}$** 用来控制锁存器是否输出数据。因为想要LED一直显示它们的状态，所以我们会要求锁存器持续地输出目前被锁存下来的数据。因此，**$\overline{\text{OE}}$** 必须一直保持低电平（接地）。此锁存器的地址解码电路相当简单。当处理器要将数据写入锁存器的时候，**A14** 为低电平、**A15** 为高电平。**A14** 经 U5A 反相后与 A15 一同连到一个与门。这个与门（U4A）的输出将会是高电平。因此，锁存器将会捕捉提供给它的数据。就效果而言，锁存器就像是大小为一个字节的存储单元。

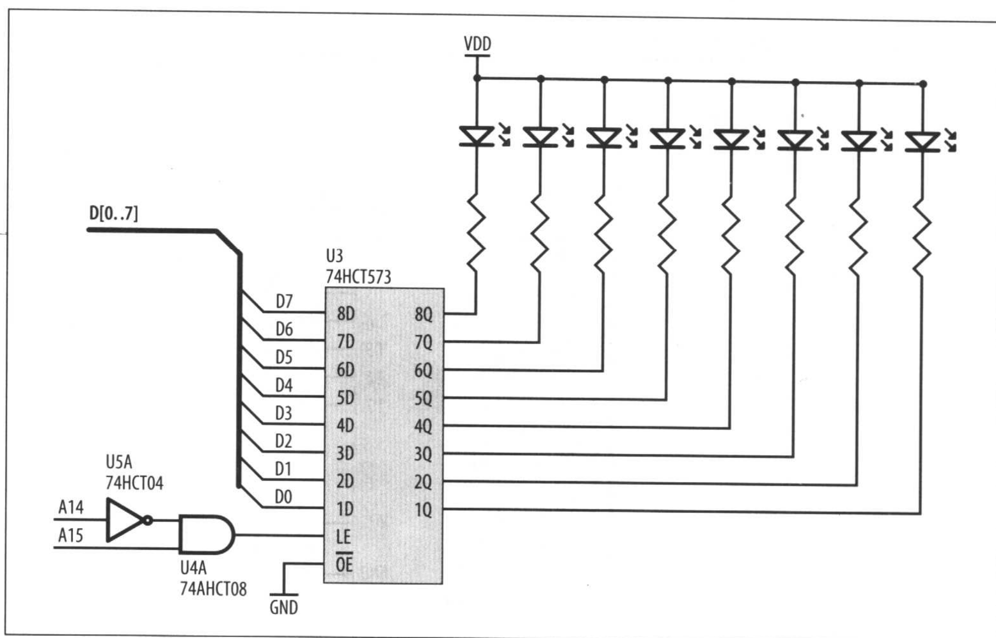


图 16-8：LED 与锁存器

图 16-9 所示为 ROM 的地址解码器。当 **A14** 和 **A15** 均为高电平时，U2D 这个与非门（NAND）的输出将会是低电平；因此，ROM 将会被选中。如果 **A14** 或 **A15** 为低电平，ROM 将不会被选中。此电路图之所以使用 32K 的 ROM（27256），而不是 16K 的 ROM，理由是 27256 比较容易获取。尽管它的容量为 32K，但是我们只会用到它的一半空间，所以地址输入端 **A14** 会被连到高电平。这样，我们将只会用到它的上半部空间。请注意，这个 **A14** 输入端与处理器的 **A14** 地址线并未连在一起。此处的 $\overline{\textbf{OE}}$ 输入端会被连到 RAM 的 $\overline{\textbf{OE}}$ 输入端。最后，在烧录期间，电源引脚 **VPP** 会被用来将数据写入芯片（来自电路外部）。正常操作期间，这个引脚会被连接到 **VCC**。

这样便完成了 68HC11 计算机的基本设计。使用处理器的 SPI 端口，还可以添加其他各种外部设备或用来存储数据的存储器。当然，使用处理器的总线通过将地址空间分配给锁存器，一样可以添加其他外部设备，并对更多的设备进行支持。我们还可以应用前一章所提到的存储管理技术来增加计算机中的 RAM 容量。

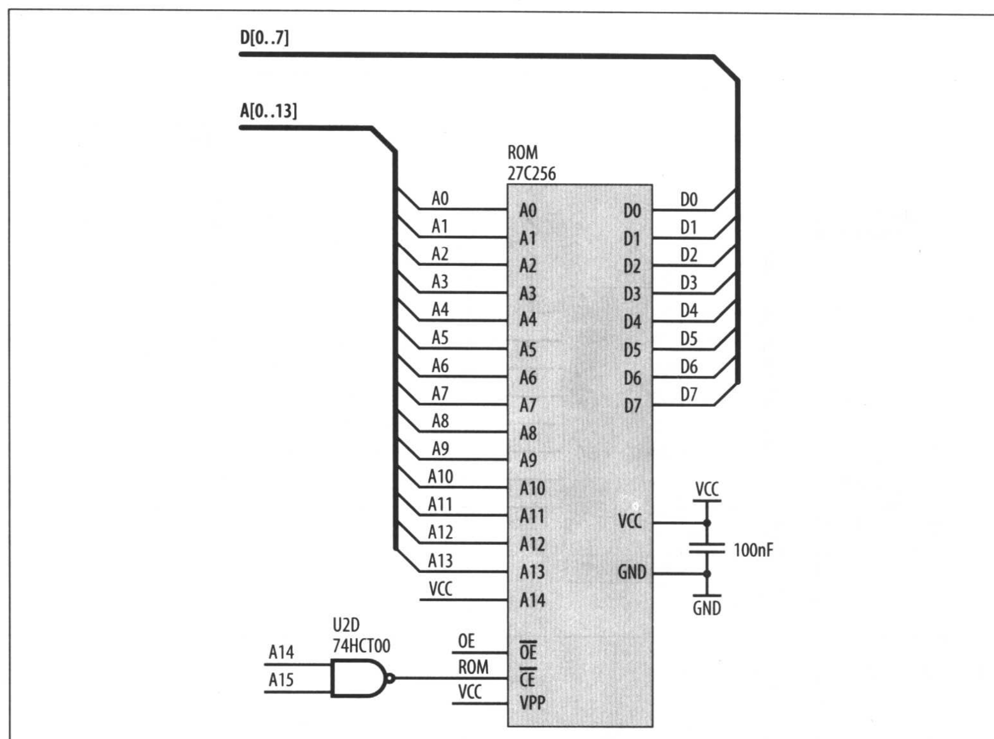


图 16-9：ROM 的地址解码电路

第 17 章

MAXQ

找不到路，我们就自己修建一条。

——汉尼拔（公元前 218 年）

本章中，我们将要看到的是 2004 年问世的新一代处理器体系结构。Maxim（*http://www.maxim-ic.com*）的子公司 Dallas Semiconductor 所开发的 16 位 MAXQ 微控制器定位于低成本、低功耗的嵌入式应用市场，该体系结构把竞争目标直接对准了 Microchip 的 PIC、Atmel 的 AVR、Texas Instruments 的 MSP430，以及许多制造商（包括 Dallas Semiconductor 自己）所提供的 8051 体系结构。对顶层的 RISC 微控制器而言，MAXQ 是一个不容忽视的竞争者。该处理器不仅速度快，具有许多功能，而且它的功耗还非常低。在撰写本书的时候，MAXQ 的使用指南已经长达 230 页。显然，这款处理器的功能非常多，有太多的内容需要说明了。因此，我只打算介绍基于 MAXQ 的系统的基本设计。首先，让我们来看看 MAXQ 何以如此的与众不同，如此让人不容忽视。

MAXQ 概览

MAXQ 所宣称的设计目标，就是要达到高度的效能功率比（performance-to-power ratio）。换言之，它的目标就是要以最低的电流达到最大的指令处理量。许多 RISC 处理器是通过指令流水线的使用来达到一个周期执行一条指令的目标。在流水线体系结构中，执行单元由许多层组成。任何一个时间点都会同时有多个指令被解码并执行。因此，尽管一个给定指令可能需要多个周期来执行，不过使用流水线之后，处理器却能够达到一个周期完成一条指令的效能（如图 17-1 所示）。

每个周期从取出指令到存回结果，都会有新的指令进入流水线。流水线的缺点是，只要遇到调用或跳转指令，就代表流水线中接下来的指令没有用了，流水线必须从其他位置

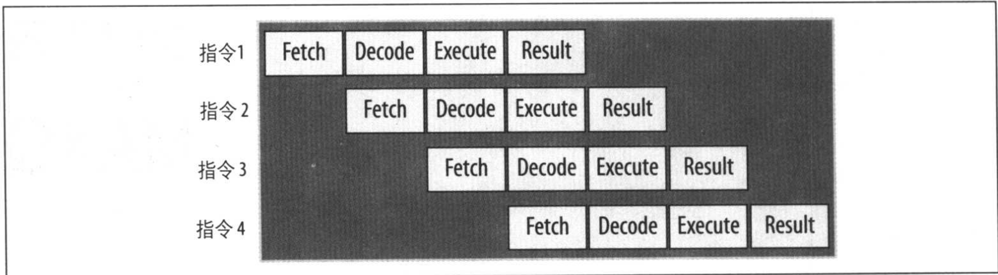


图 17-1：四级指令流水线

（跳转/调用指令所指向的位置）重新载入指令。所以，尽管流水线可以达到一个周期执行一条指令的效能，但是它也让效能大打折扣，除非程序的结构是线性的。

MAXQ 并不具备指令流水线，但是通过 long jump 和 call 指令的例外处理，以及若干经扩展的存储器存取，它仍旧可以达到一个周期执行一条指令的效能。你可能会说“这有什么不同？”，流水线式的处理器也同样有 jump 和 call 的问题。两者的差别在于，流水线式的处理器执行 jump 指令时，不仅 jump 指令无法在一个周期内完成，而且也会因打乱了指令的执行流程而影响到流水线中接下来的其他指令。MAXQ 就不同。只有 jump 或 call 指令无法一个周期内完成，其后的指令的执行并不会因流水线重新载入其他指令而有所延迟。

MAXQ之所以会有这么好的效能，是因为它的指令解码与执行单元相对于其他处理器而言简单许多。有多简单呢？事实上，MAXQ 只有一条指令（`move`），但是这条指令却有许多功能，具体功能取决于目的操作数。因为只有一条指令，所以不需要用典型的解码单元（既然已经知道这条指令是什么了，何必要进行译码呢？）。因此，指令的执行就变成了判断来源和目的，以及是否有额外的硬件操作会被触发成 move 的一部分。指令里的来源与目的字节只会对内部的数据路径起作用，而这发生在指令被取出来的时候。

MAXQ 指令的基本格式为“`fdd dddd ssss ssss`”，其中 `f` 是格式位，`d` 代表目的操作数，而 `s` 代表来源操作数。当格式位是 1 时，指令会将数据从一个索引块移动到另外一个。当格式位为 0 时，会有一个 8 位的立即数被载入到索引模块（见表 17-1 所示）。

表 17-1　MAXQ 的指令格式

格式位	7 位的目的操作数	8 位的源操作数
1	索引模块	索引模块
0	索引模块	立即数数据

在MAXQ体系结构中，索引块并不必一定就是特殊寄存器，因此这个体系结构很有弹性。一个累加器实际上可以代表一个以上的索引块。一个索引块将能够以累加器为目标，并且执行加法操作；然而另一个索引块也能够以相同的累加器为目标，不过所进行的却是减法操作。因此，两个索引块可指定两种不同的操作，即使它们的目标是同一个寄存器。很聪明，不是吗？

注意：几年前，我设计了一个实验性的零操作数处理器（作为soft core之用），也实现了不使用流水线的单周期执行（一个周期执行一条指令）功能。这个零操作数处理器的寄存器被设置成堆栈，所有的指令都会自动在堆栈上进行操作。例如，add指令将会从堆栈的栈顶取出两个数据（寄存器r0和r1），把它们加在一起，最后把结果存回到堆栈（r0）的栈顶。因为源操作数和目的操作数总是已知，所以运算单元可以预先进入ALU，它会以平行的方式计算所有可能的操作数。接着进来的指令只需要选择适当的ALU结果或操作。在许多方面，这个概念如此简单，足以媲美MAXQ，但是它与MAXQ之间就像是同一硬币的两面一样。MAXQ虽然只有一条指令，不过它却具有很多索引块；而零操作数处理器虽然有许多条指令，但是它只有一个源操作数和目的操作数。

欲了解以堆栈为基础的语言，请参考第3章的相关内容。

如同PIC处理器和AVR处理器一样，MAXQ也是一个采用哈佛体系结构的处理器，有单独的代码空间和数据空间。总而言之，MAXQ是一个非常棒的处理器，我确信它日后必定会在市场上占有一席之地。

有了上述了解之后，让我们来看一下如何设计一个简单的基于MAXQ的嵌入式计算机。正如你将看到的那样，尽管MAXQ是一个功能强大的处理器，但当你在遇到设计上的“难题”时，你所能做的也很有限！

电路原理图

在信号混杂的环境（同时包含数字元件和模拟元件）中使用微控制器，常会遇到噪声干扰数字子系统的问题。较高效能的处理器一般会在模拟部分产生更大的噪声，除非你在这方面下过苦功，才有办法将这些效应降至最低。因此，要达到较高的效能，通常也意味着你必须努力除去模拟电路中所产生的噪声。MAXQ所提供的智能时钟管理功能，可通过只提供时钟给有需要的子系统（并且只在它们需要的时候）来降低噪声。因此，整体的数字噪声会大幅降低。MAXQ处理器需要两个晶体：一个是给主CPU时钟使用的16MHz晶体（X1），一个是给定时器使用的2.768kHz手表用晶体（X2）。图17-2所示为MAXQ2000F处理器与其支持元件。

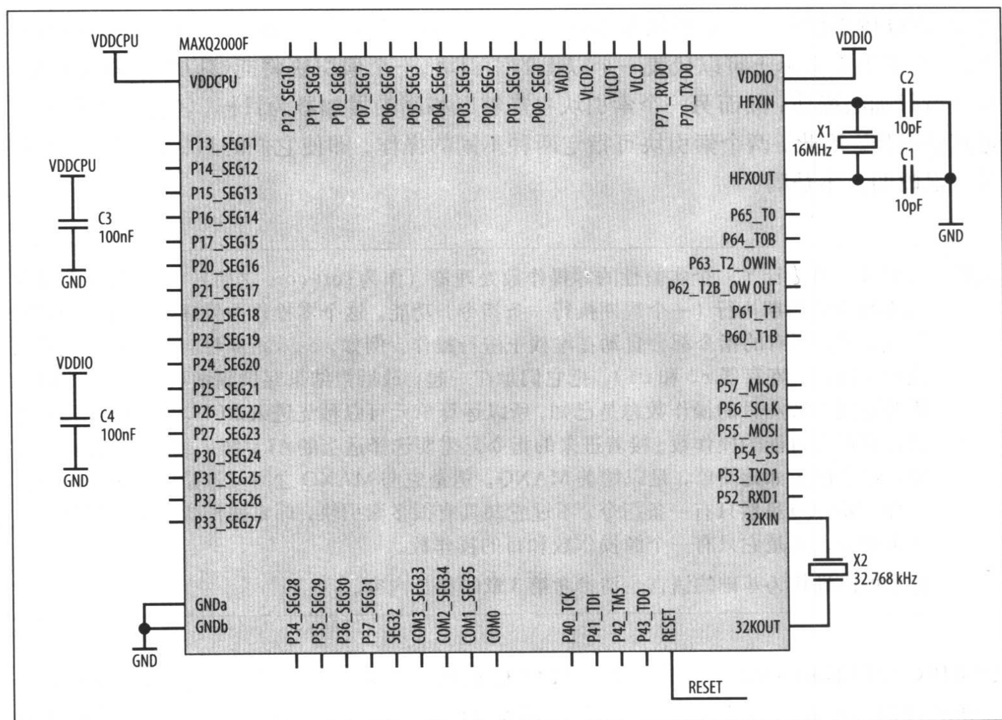


图 17-2：MAXQ 处理器与支持元件

MAXQ 处理器需要两种电源：**VDDCPU**（2.5V）和 **VDDIO**（3.6V）。它们都需要经由 100nF的陶瓷电容接地，以便进行退耦。我们可以使用MAX1658来产生2.5V的电源（如图 17-3 所示）。MAX1658 是一个通用的稳压器，它的输出经由偏压电阻来进行调整。此例中，我们通过偏压电阻 R1 和 R2 将输出设置成 +2.5V。偏压电阻的精确度很重要，所以我们选择具有 1% 容差的电阻。稳压器的输入和输出都需要经过 10μF 的电容进行退耦。MAX1658 可以运行在 2.7V~16.5V 的输入电压（**VIN**）下，至多可以为嵌入式计算机提供 350mA 的电流。

类似的电路可用来产生 3.6V 的电压供 MAXQ 的 I/O 子系统使用（如图 17-4 所示）。注意，要产生 3.6V 而非 2.5V 电压，所采用的电阻值就会不同。

MAXQ内部集成了上电复位发生器，可以在上电时将处理器设置为一个已知状态。不需要外部的复位电路。如果你需要手动复位，可以使用一个按钮开关来将 **RESET** 引脚置为低电平。然而，请特别注意，**RESET** 是一个双向引脚，MAXQ 也会使用此引脚输出信号，以指示目前正在进行的复位（可能产生自内部）的处理。（如果有需要）系统设计者也可以利用此信号来复位外部的设备。

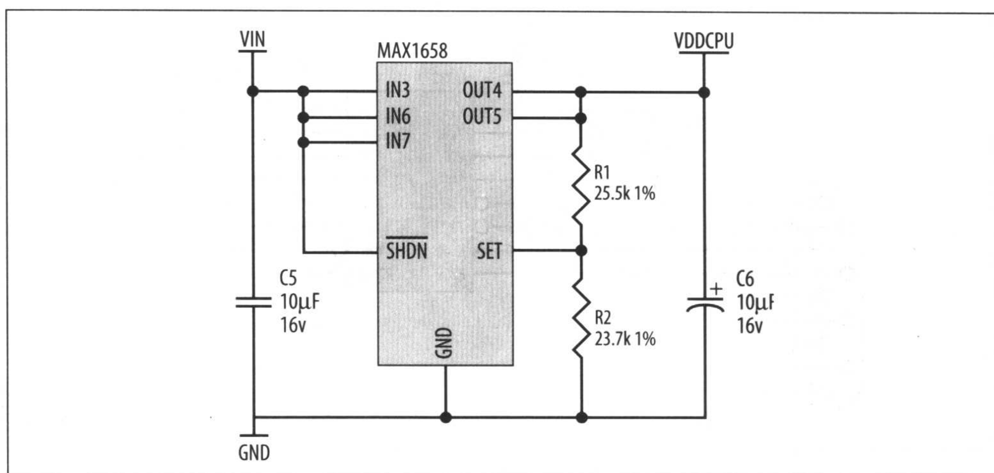


图 17-3：产生 VDDCPU 所用的 2.5V 电压

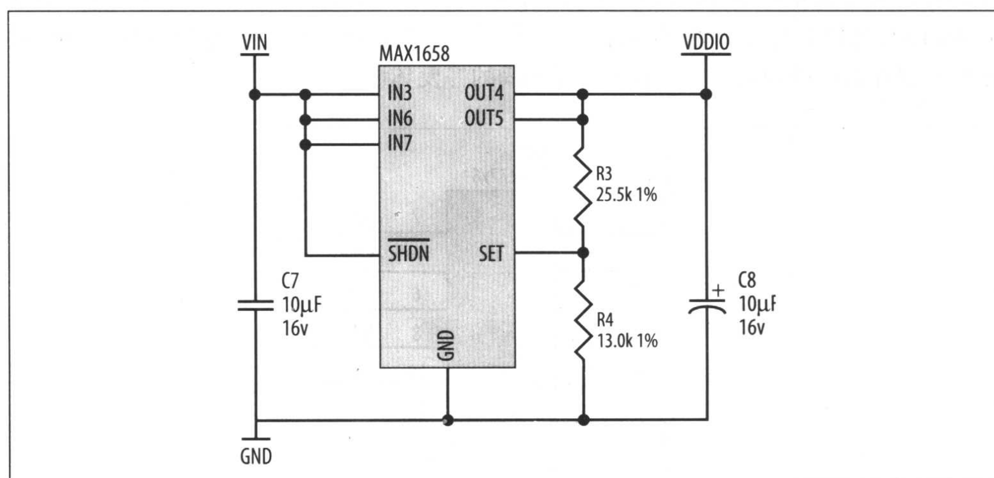


图 17-4：产生 VDDIO 所用的 3.6V 电压

被标识成Px的每个端口，可用来存取MAXQ的I/O。除了提供数字I/O功能，它们还有双重用途。如端口5除了可用作一个SPI接口，也可用作一个串口。SPI接口可用来连接任何基于SPI接口的外设。串口则需要一个电平转换器，如MAX3232，如图17-5所示。MAXQ的发送器（TXD1）和接收器（RXD1）引脚会被连接到MAX3232的接收器和发送器引脚。

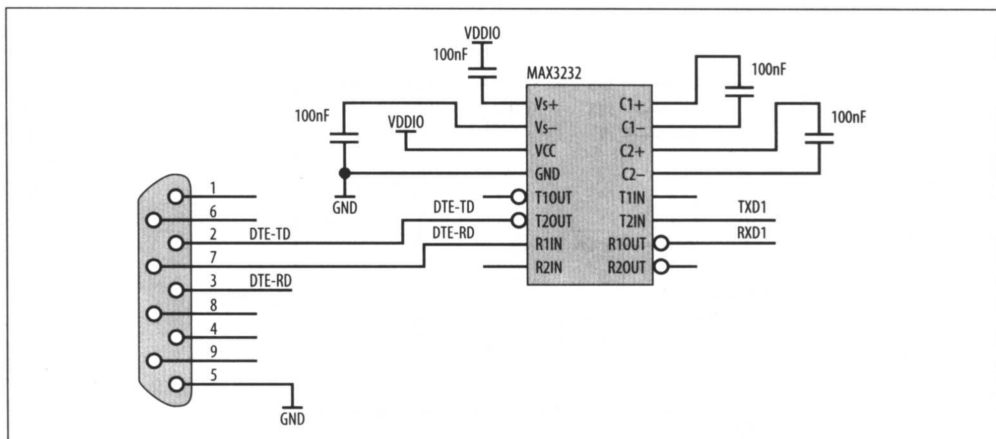


图 17-5：串口

在进行在系统编程和调试时，MAXQ的端口5用于提供对JTAG信号的访问。图17-6所示为JTAG连接器的引脚图。此引脚图与Maxim的MAXQ开发系统上所使用的一样，这使你可以在自己的嵌入式计算机中使用相同的开发环境。

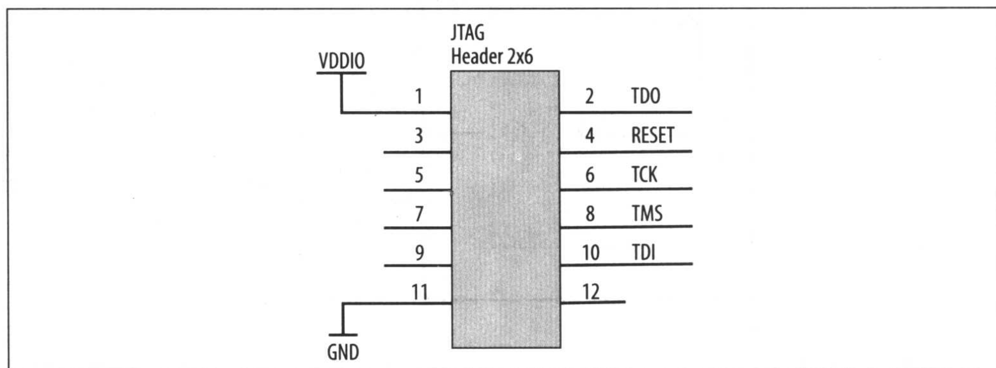


图 17-6：JTAG 接口

端口1、2和3还可以作为一个36段（数字型）LCD的接口。图17-7所示为连接到LCD带状电缆的连接器。

MAXQ是一个速度快、功能多的16位处理器，这要归功于Maxim对它所进行的扩充。如果你正在为嵌入式应用寻找一个低功耗却多功能的处理器，那么你可以仔细地评估一下MAXQ，你将会发现，MAXQ是一个令人印象深刻的小型处理器。

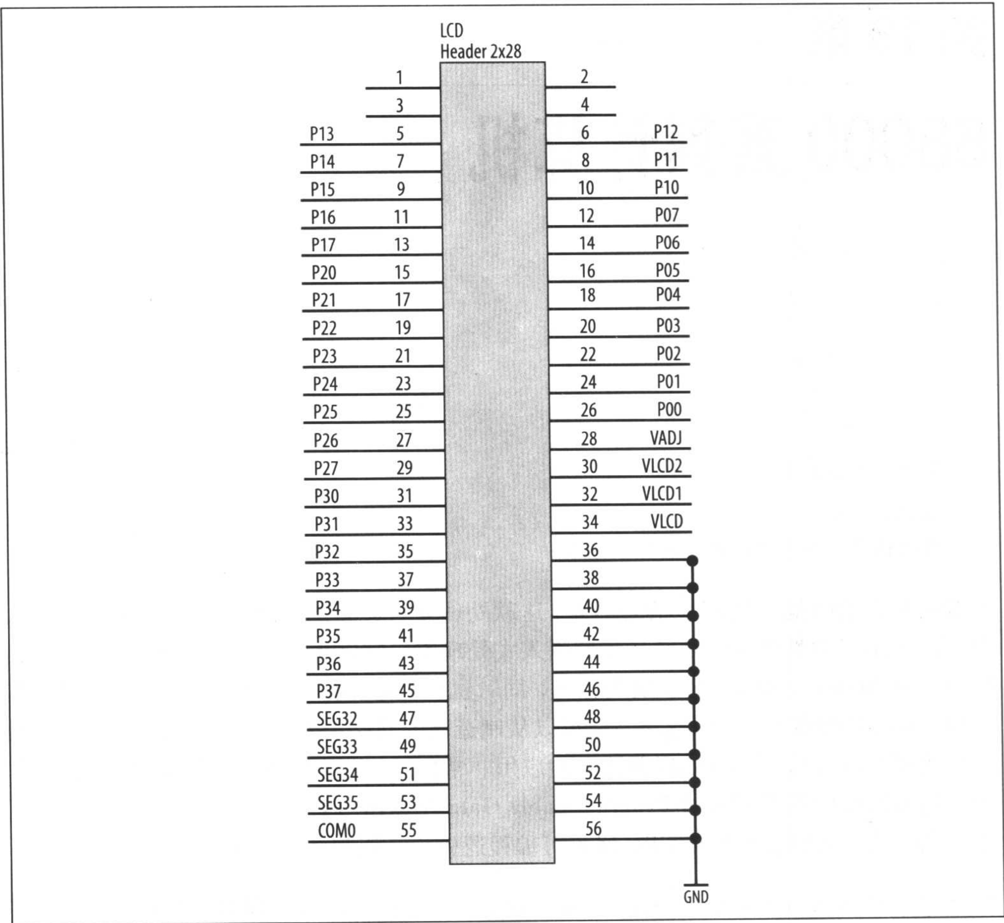


图 17-7：数字型 LCD 接口

第18章

68000系列计算机

一切皆流，万物未曾停滞。

——赫拉克利特

第欧根尼·拉尔修所编《名哲言行录》

本章所要介绍的是一款32位的处理器，这款处理器从其面世至今已有很长一段时间了，由它演化出了种类繁多的微控制器和嵌入式处理器。68000处理器（也被称为68k）由Freescale Semiconductor公司负责生产，同时也授权给其他的一些制造商共同生产。基于68000的处理器的种类非常多（你可以从制造商的网站上得到处理器的名称列表以及它们的相关特性）。68000处理器适合的应用非常广泛。你甚至还可以将68000处理器作为现场可编程门阵列（Field-Programmable Gate Arrays，FPGA）的软内核（soft core），这就意味着你可以把68000 CPU嵌入可编程逻辑中，以形成一个单一集成芯片。

1979年，Motorala Semiconductor公司（注1）在其8位6800系列芯片的基础上开发了MC68000处理器。这种新型的处理器具有以下的典型特征：更大的地址空间、32位的寄存器、多种寻址模式以及一个扩展指令集（包括超过一千条的操作码）。它设计的目的是为了在通用或专用的Unix操作系统中执行多任务操作系统。到现在为止，它已在Unix机器上工作了很长的时间，不过目前在这个领域却已被更先进的RISC机器所吞噬。68000处理器也曾应用于一款早期的Macintosh机，以及Atari ST、Commodore Amiga和Jef Raskin的CAT计算机中。不过，所有这些计算机都已经成为历史。

注1：　Motorala Semiconductor即现在的Freescale Semiconductor公司。

注意：如果你想获取有关CAT的有趣综述，可以读一读Jef Raskin的《The Humane Interface》一书（Adalison-Wesley出版）。他在本书中讨论了CAT的独特设计，对用户接口设计也有一些有趣的观点。

由于68000处理器支持大量的软件和合理的计算功率，所以目前它被广泛应用于嵌入式系统中。目前它已经发展成为一系列应用于嵌入式系统、工业控制、网络和PDA的微控制器，而其中的683xx系列处理器则是为了嵌入式应用而专门设计制造的。这些处理器把多种功能（如UART、SPI、ADC等）都集成到一个CPU32内核（基于68020）中。另外，还有一些为特殊应用开发的68000系列的处理器。掌上型PDA中的68EZ328 DragonBall处理器也是在CPU32内核的基础上，再辅以具有一系列PDA常用功能的LCD控制器而开发出来的。而DragonBall芯片再加上内存实际上就构成了一个PDA，而且ucLinux也在它们的小型的嵌入式控制面板上使用了DragonBall处理器。

ColdFire系列处理器是68000采用RISC技术后的升级产品，现在被广泛应用于工业控制以及网络接口设备上。

68000系列处理器是一款性能优良的通用处理器。它们具有非常好的指令集，使得为其编写代码变得容易（而有趣），因而用它们构建计算机系统也变得相对简单。这一系列的处理器都具有很大的地址空间，而且还支持异步操作，这就使它们能够连接多种不同操作速率的存储器和外围设备。它们主要用在工业控制和监控领域以及一些家用电器中。

在这一章，我们将要看到的是标准68000处理器。不过在实际的设计中，你可能不会使用它，因为基于68000的集成控制器很可能会更适合你的需求。那么，我们为什么讨论它而不是其他的同类产品呢？首先，基于68000系列的处理器种类太多。其次，所有这些处理器都基于68000，所以掌握基本的68000处理器是一个很好的入手点。最后，因为所有的处理器的变种都要比它们的祖先更容易使用，所以如果你知道如何使用标准68000进行设计，那么你也就会用其变种来进行设计了。

可以通过学习68000处理器去了解更多的处理器，包括一些公用的编译器。68000系列可以使用很多商用的C编译器和汇编器。另外，68000还被GNU开发套件完全支持，可以像使用其他的商用操作系统那样使用Linux和BSD。

68000处理器的体系结构

68000有8个32位的数据寄存器（D0~D7）、8个32位的地址寄存器（A0~A7）、一个32位的程序计数器、两个32位的堆栈指针以及一个16位的状态寄存器（如图18-1所示）。68000处理器可以处理的数据包括32位字、16位字、字节（byte）或位（bit）。

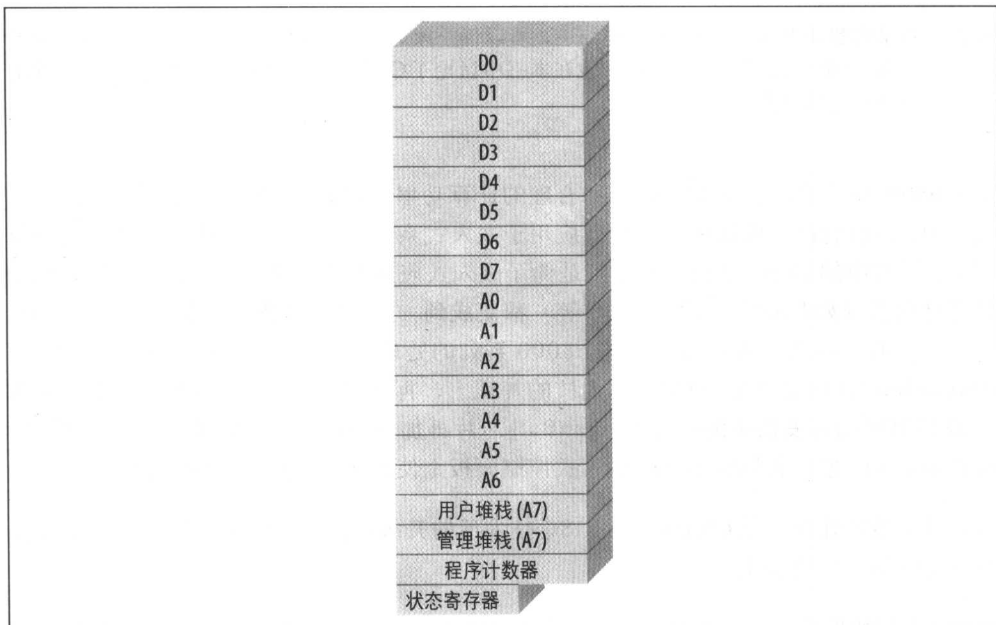


图 18-1：68000 的程序设计模式

处理器有两种操作模式：管理模式（操作系统）和用户模式（应用程序）。由于操作模式对外部硬件也是可见的，因而允许地址解码器有独立的管理空间和用户空间。

标准68000芯片也是一款传统的基于总线的处理器。图18-2所示是68000系列处理器的结构框图，图中也给出了其中一款处理器的引脚示意图。68000系列的引脚和信号也许都不完全一样，但是它们都具有相同的核心功能。连带附加的I/O能力，嵌入式控制器也把这些基本功能模块添加进来。下面我们就快速地浏览一下MC68EC000处理器的各个引脚功能。你也可以从Freescale Semiconductor的网站(*http://e-www.Freescale.com*)上下载68000系列的技术手册、程序员手册以及技术参考。当你登录公司的网站后，可以查看Freescale系列处理器，从小型的8位控制器到64位的PowerPC RISC处理器，你想得到的信息都在里面。

最初的68000处理器有一条23位的地址总线（A1~A23），使它可以访问16M的内存空间，另外它还包括一条16位的数据总线。而大部分其他的基于68000结构的处理器都有32位的数据和地址总线，这使得它们最大能够访问4G的内存空间。

68000还有一个用以驱动所有处理器操作的输入时钟。如果没有使用等待状态，那么存储器访问的典型时间为8个输入时钟周期。另外，68000系列的处理器还有一个内置的地址解码器以及一个可通过软件配置的等待状态产生器，使得连接更为简单。

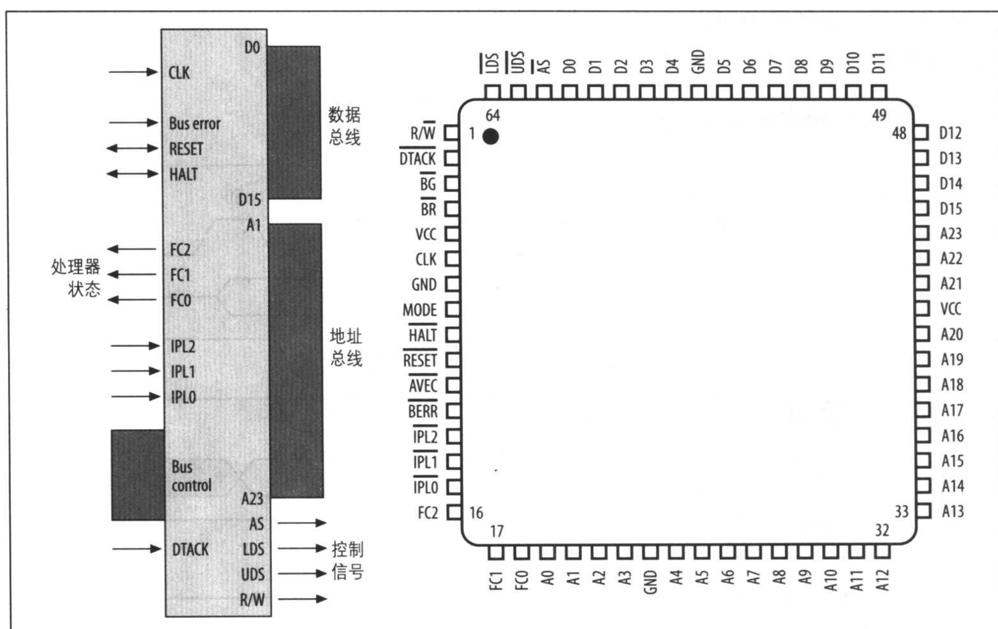


图 18-2：MC68000 结构和引脚图

处理器具有一个地址选通 **AS**（address strobe）引脚，用来标识已有一个有效的地址被发送到地址总线上。同样，数据选通（**LDS**、**UDS**）引脚用来表明一个有效的数据在数据总线上，另外 **R/W** 引脚被用来判定传输的方向。而数据传输确认输入引脚 **DTACK**（Data Transfer Acknowledge）则允许外部设备向处理器表明它可以终止当前的存储周期（在某些处理器中称之为数据传输确认）**DTACK**。功能代码输出（**FC0**、**FC1** 和 **FC2**）用来表明当前处理器的操作模式（管理态或用户态）。总线错误 **BERR**（Buss Error）被外部地址解码器用来表示一个错误状态。它可以帮助系统跳出内存中未使用的区域，而且还能在状态信号线的协助下检测用户是否非法访问了管理态专用的内存区域。例如，如果一个程序出现故障，并且在此期间试图访问那些未被划分给外设的内存区域，这时地址解码器就会给处理器发送错误返回信号，对 **BERR** 信号的确认会使处理器执行中断指令，并采取适当的操作。**HALT** 引脚被用来使处理器挂起，而无需产生复位命令。三个中断输入（**IPL0**、**IPL1** 和 **IPL2**）被用来产生七级中断处理。总线赋予 **BG**（Bus Grant）和总线请求 **BR**（Bus Request）分别用作 DMA 的控制信号，这样可以使其他的处理器继续使用计算机的总线。还有 MODE 引脚，它只被部分处理器使用，用以判断和决定 68000 是使用 16 位的还是 8 位的数据总线。MODE 作为来自复位操作的抽样。**AVEC** 引脚也只是出现在一部分 68000 系列处理器中，用以判断处理器是否对中断采取自动检测方式。如果自动检测被置位，外部设备将提供适当的中断向量。这样，发出一个特定的

中断信号后，外部设备可以指定处理器要执行的操作类型。其他的68000系列机可能还有其他的信号，但是上述的是其中最主要的。

图 18-3 所示是 68000 存储访问的基本时序图。

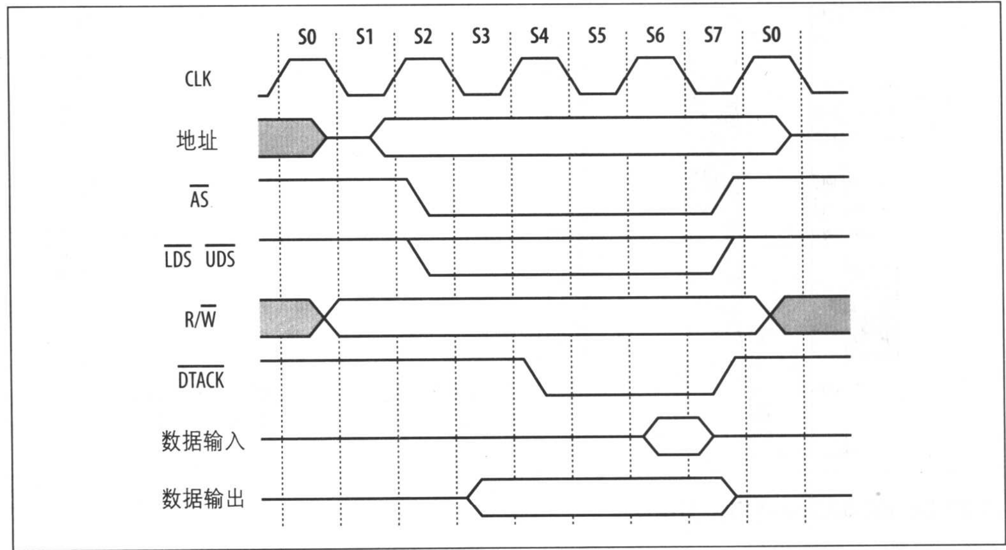


图 18-3：MC68000 时序图

68000的存储周期被分成若干时钟状态（S0~S7），其中 S0 是一个存储周期的开始。在存储周期开始前，处理器需要确定读/写状态，其中发送低电平表示一个写访问周期。高电平表示读访问周期。而处理器发出其他的状态信号时，表示地址总线处于从前一次内存访问到下一次访问的过渡态。到S2状态时，处理器将输出有效的地址，同时将地址选通AS，置为低电平，用以表示有效地址已准备就绪。低数据选通**LDS**(lower data strobe)和高数据选通 **UDS**（upper data strobe）在合适时也变为低电平，用以计算存储访问的时间宽度。对一个 16 位的数据传输来说，**LDS** 和 **UDS** 都会响应。而如果是一个 8 位的数据传输，那么只有 **LDS** 和 **UDS** 中的一个作出响应，具体取决于当前传输的是高位字节还是低位字节。如果进行的是写存储访问周期，那么处理器将在 S3 状态时输出有效数据。这时，所有来自处理器的输出都是有效的，然后处理器等待被访问的设备作出响应。

在 S4 状态的时钟下降沿，处理器开始检测数据传输确认（**DTACK**）输入引脚的状态。如果 **DTACK** 为高电平，那么处理器将插入等待状态，继续重复检测直至在某个时钟周期的下降沿 **DTACK** 及为低电平（我们将会在后续的章节中讨论如何产生等待状态）。当 **DTACK** 处于低电平时，处理器将会确认它，以便待访问的外设有足够的时间来响应和

终止此次访问周期。如果为读周期，则处理器将会在 S6 状态内的时钟下降沿将数据锁存。如果为写周期的话，被访问的外设将在 S7 状态内且数据选通信号升高时锁存数据。

在早期的6800系列处理器中还提供了对同步操作的支持，它们是通过使用控制信号来实现的。不过，6800 系列早已成为历史，而且与它们兼容的外设现在也十分少见，所以我们就暂且不管6800的控制信号。需要指出的是，目前大部分基于68000的处理器都不再为 6800 的外设提供支持。

简单的基于 68000 的计算机

现在我们要看的是一款小型的基于68000的计算机。为了简单起见，我们假设它只有很少量的存储空间，而且只有一个外设，即 MK68901 多功能外设（Multi-Function Peripheral，MFP）由 ST 电子设备公司生产。MFP 带有一个 UART、并行 I/O 及中断控制。图 18-4 所示为这个系统的结构框图。

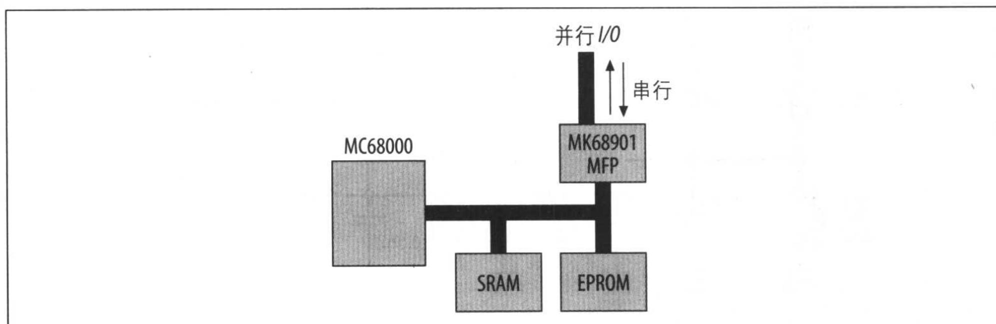


图 18-4：基于 68000 的计算机

在设计这个系统时我们只用了少量的内存，目的就是避免设计的复杂化。虽然它还远逊于大多数的台式机，但对很多小型控制应用来说已经足够了。这个设计可以用来实现一些小型的应用。MK68901的计数器可以用来监控外部事件脉冲或者产生PWM信号用以控制电机。这种计算机可以通过它们的串行端口来接收命令，并且通过使用 MK68901 上的并行 I/O 引脚来激活（或挂起）外部子系统。这个基本的设计通过部分的修改，还可作为RS-232C接口与并行端口间的桥接器。比如，你可以用它来实现一个只有串口的计算机和并行端口打印机之间的连接。当然，你还可以把一个串口的调制解调器连接到你的PC机的并行端口。使用我们在 AVR 一章中介绍过的总线接口技术，还可以给系统增加更多的外设，例如模数转换器 ADC 和数模转换器 DAC、以太网或其余更多种的外设。可以说它可能完成的应用是很难穷尽的，不过这一切都还得从下面的核心设计做起。

所以，下面就让我们开始基于68000计算机系统的设计吧。我们将会依次看到复位电路、地址解码器、I/O 和存储器。

复位电路

为了使MC68000复位，**HALT**和**RESET**都必须同时置为低电平。又由于这两个信号线都可以作为处理器的输出端，因而它们都必须由复位电路通过开断集合器门独立驱动。68000的传统复位电路如图 18-5 所示。

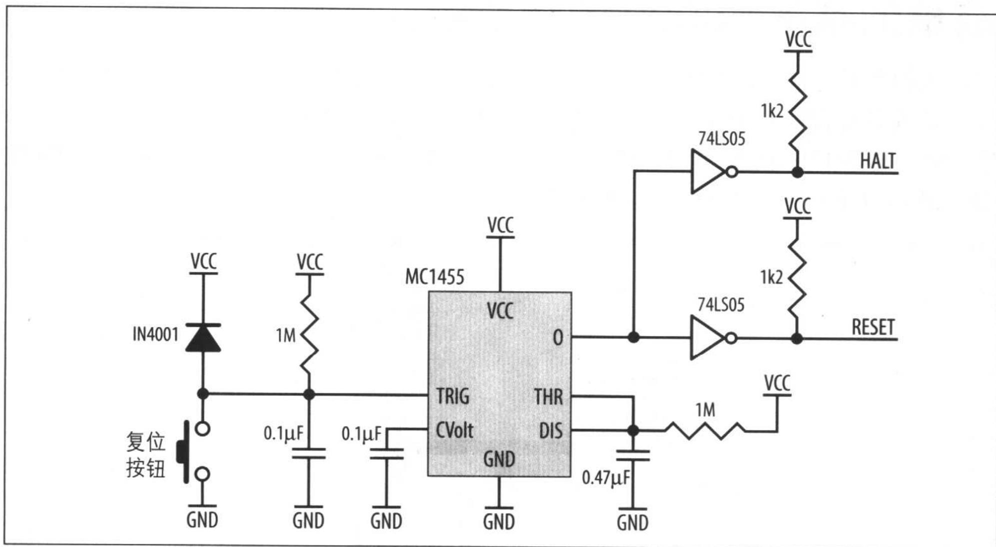


图 18-5：复位电路

MC1455通过将其输出端置为低电平的方式来响应VCC中断，而输出被用来将**RESET**和**HALT**同时置为低电平。除非复位开关被按下，否则在正常情况下**RESET**将由上拉电阻使其一直保持为高电平。此时，二极管被用来消除那些可能会给**VCC**发送TRIG的毛刺信号。

图 18-6 所示的是一个更好的复位电路，它使用了集成复位控制器的MAX825 芯片。**RESET** 和 **HALT** 都需要低电平驱动。

地址解码器

在68000处理器中，PAL逻辑电路被用来完成地址译码以及分别产生读/写选通信号的功能。通常，处理器的地址选通信号**AS**用来标识一个已经被发送到总线上的有效的地址。地址解码表达式如下所示：

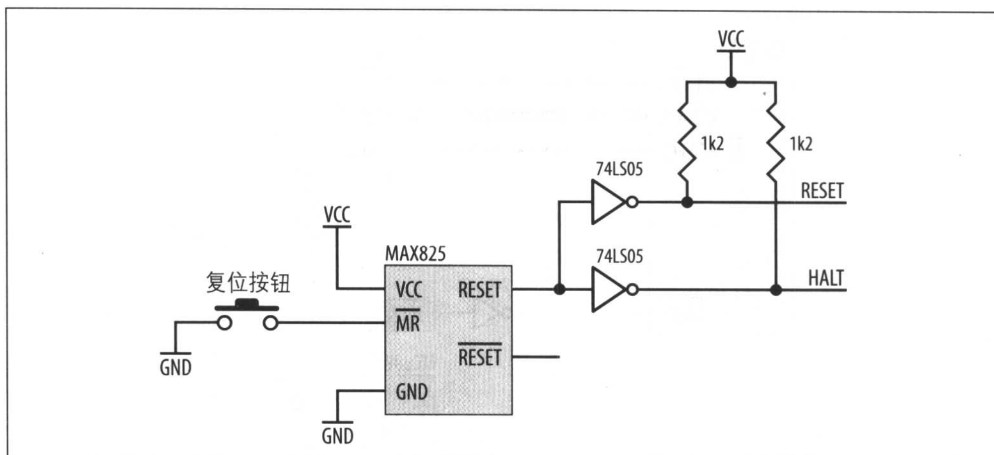


图 18-6：68000 的 MAX825 复位电路

```
ROM = /(/AS * /A23 * /A22)
RAM0 = /(/AS * /A23 * A22 * /LDS)
RAM1 = /(/AS * /A23 * A22 * /UDS)
MFP = /(/AS * A23 * /A22)
```

不过 MFP 是个例外，它可以产生自己的数据传输确认信号 **DTACK**，**DTACK** 可以作为所有其他设备地址解码的一部分。由于来自 PAL 的 **DTACK** 信号和来自 MFP 的 **DTACK** 信号必须执行 OR 操作，所以它必须由一个开断集合器门驱动。

用来产生 **TACK** 的 PAL 表达式很简单：

```
TACK = (/AS * MFP)
```

因而，只要处理器访问它的地址空间 **TACK** 就会被激活（高电平），不过前提是它没有访问 MFP。从表达式可以看出，如果地址选通信号 **AS** 为高电平，或者访问了 MFP，那么 **TACK** 都将为低电平。PAL 的 **TACK** 输出信号通过一个开断集合器 74LS05 反转，并与来自 MFP 的 **DTACK** 信号执行 OR 操作（直接连在一起）。由于这个输入将会有很大的电流产生，所以 **DTACK** 需要一个 1kΩ 的上拉电阻，如图 18-7 所示。

之所以没有为 **BERR**（Bus Error）信号的产生作准备，是因为简单的地址解码逻辑已经分配了所有的地址空间。不过如果我们还有一些未用的内存空间，那么当访问这些未用的地址空间时，就需要使用地址译码器来产生 **BERR** 信号。

PAL 产生的用于存储芯片的独立的读 / 写选通信号的表达式如下所示：

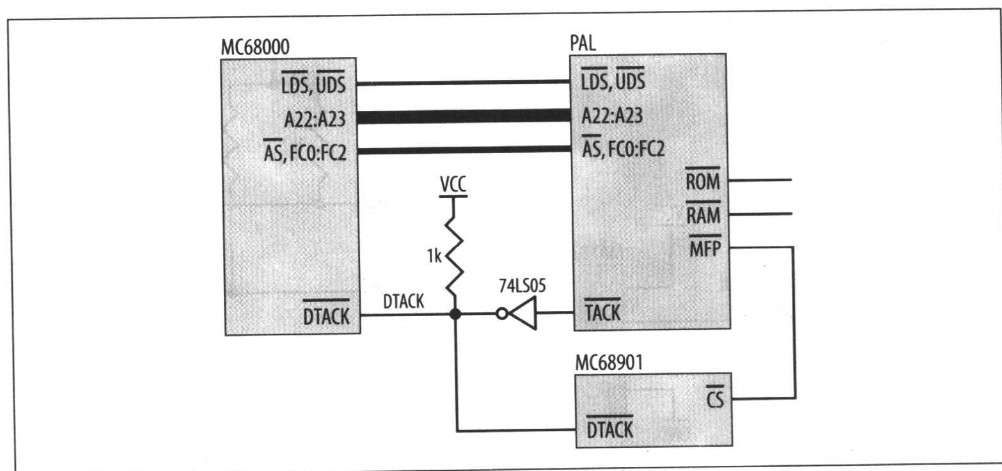


图 18-7：地址解码器产生 DTACK

```
UWE = /(/UDS * RW)
LWE = /(/LDS * RW)
UOE = /(/UDS * /RW)
LOE = /(/LDS * /RW)
```

图 18-8 所示的是 PAL 的连接电路。PAL 还带有额外的地址空间以防将来存储映射会有变化。PAL 使用处理器的时钟（**CLK**）来产生 MFP 的时钟信号 **MFPCLK**。

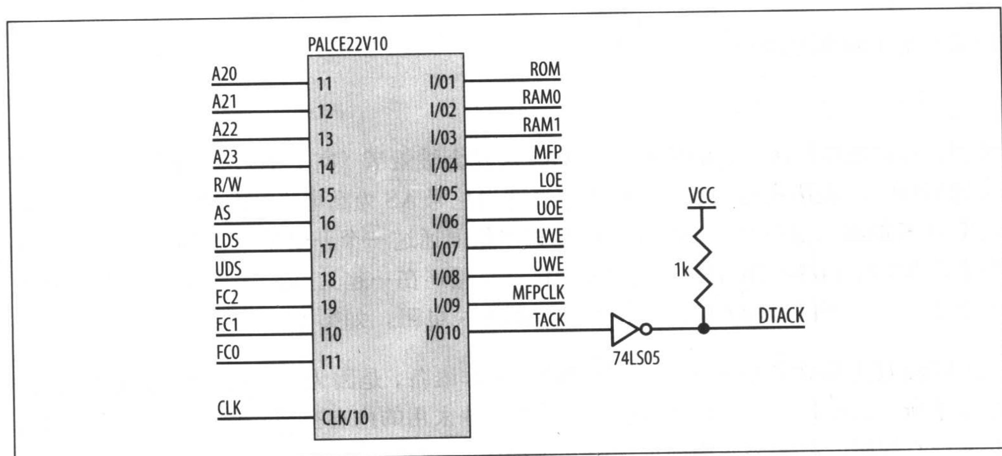


图 18-8：地址译码器和系统逻辑电路 PAL

功能代码输出（**FC0~FC2**）可以由 74LS138 解码，用以驱动三个 LED 灯（如图 18-9 所

示)，它们可以提供一个可见的处理器状态标识。而且如果你需要独立的管理地址空间和用户地址空间的话，那么地址解码器也可以使用功能代码。对一些更成熟的外围设备（如 MFP）来说，如果它们产生一个中断，就需要处理器去确认。由于这些功能代码还能够标识一个中断确认信号 IACK（Interrupt Acknowledge），所以 74LS138 也使用这些功能代码产生外设的 IACK。

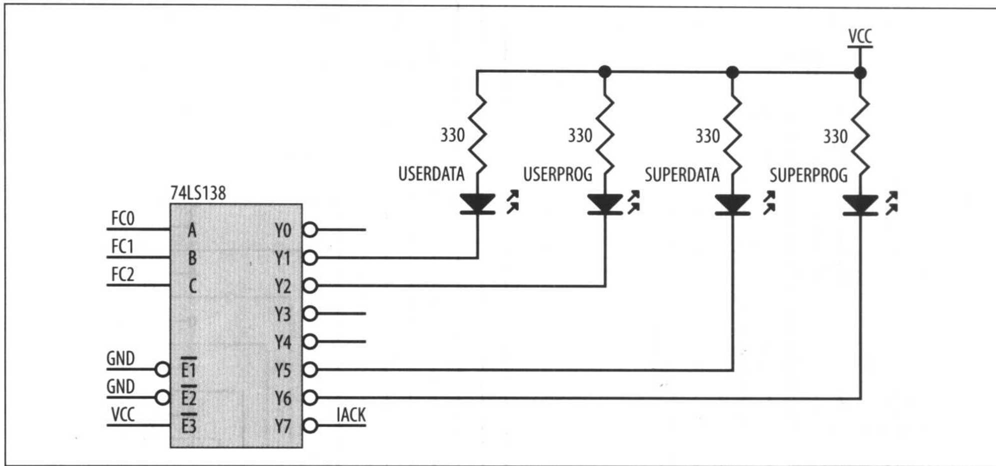


图 18-9：用以标识处理器模式的状态 LED

输入 / 输出

MK68901 MFP 提供了一个串口，以及一些并行 I/O 功能、16 个中断源控制器和 4 个 8 位的定时器。MK68901 还内置了一个用以驱动内部定时器的振荡器。时钟输出端 TD0 将时钟信号反馈给 MFK，用作串行接口的时钟。因而内部振荡器必须以合适的频率运行，以配合RS-232的工作，因此振荡器在一个频率为3.6864MHz的晶体的控制下工作，这个频率可以被 MFP 划分，以给串行端口提供合适的波特率。MAX3232 电平转换器将来自 MFP 的串行总线的电平转换为 RS-232C 的电平。一个 9 引脚的 D 型触发器用于访问 RS-232C 信号。并行 I/O 线和定时器输入及输出通过一个 26 引脚的 IDC 连接器起作用。

MFP 的电路原理图如图 18-10 所示。

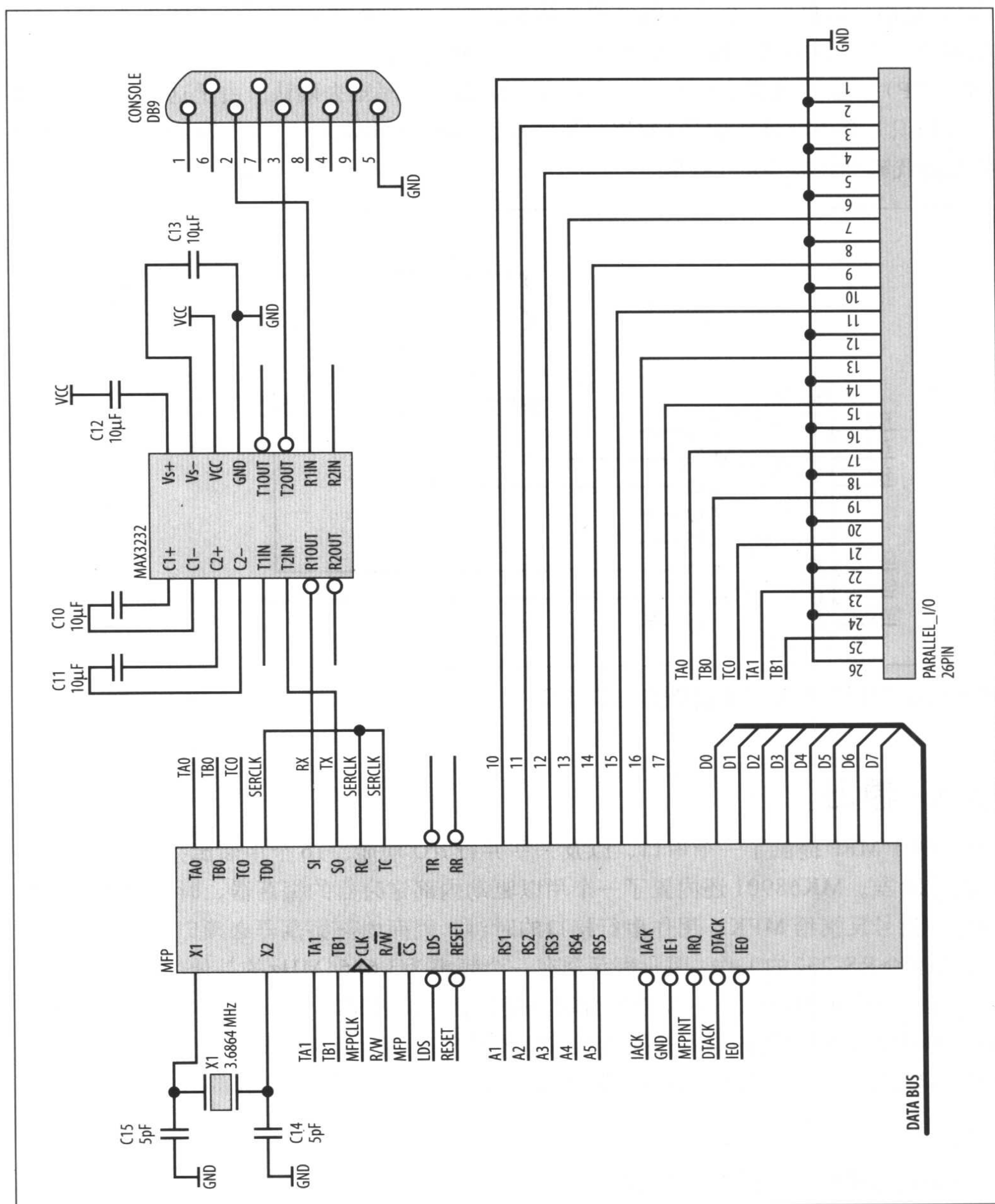

CONSOLE
DB9
GND
C13
10µF
VCC
GND
VCC
C12
10µF
MAX3232
Vs+
Vs–
VCC
GND
T1OUT
T2OUT
R1IN
R2IN
C1+
C1–
C2+
C2–
T1IN
T2IN
R1OUT
R2OUT
C10
10µF
C11
10µF
PARALLEL_I/O
26PIN
TA0
TB0
TC0
TA1
TB1
SERCLK
RX
TX
SERCLK
SERCLK
MFP
TA0
TB0
TC0
TD0
SI
SO
RC
TC
TR
RR
X1
X2
TA1
TB1
CLK
R/W
CS
LDS
RESET
RS1
RS2
RS3
RS4
RS5
IACK
IE1
IRQ
DTACK
IE0
D0
D1
D2
D3
D4
D5
D6
D7
MFPCLK
MFP
A1
A2
A3
A4
A5
GND
MFPINT
DATA BUS
X1
3.6864 MHz
C15
5pF
C14
5pF

图 18-10：多功能外设

存储器

系统使用了一个 256K 的 EPROM 和一个 512K 的 SRAM，SRAM 的连接如图 18-11 所示。注意，由于68000的数据总线的宽度为16位，所以需要两个SRAM。而对于其他一些使用32位外部数据总线的基于68000的处理器来说，则需要并行使用4个存储芯片。另外，还需要注意数据总线的每一半部分是如何分别连接到两个芯片之上的。

现在看一下地址线和SRAM的连接。处理器的最低地址位 **A1** 被连接到 SRAM 的 **A0** 输入端，依次类推。由于处理器访问外部存储器是以16位字为单位，所以 **A1** 表示最低地址有效位。换句话说，当你逐字地往存储器中存放时，**A1** 将会首先变化。而SRAM的最低有效位是 **A0**，不过因为是两个SRAM联合表示一个16位的存储字，所以SRAM的 **A0** 端必须连接处理器的 **A1** 端。其余的地址位则依次连接。

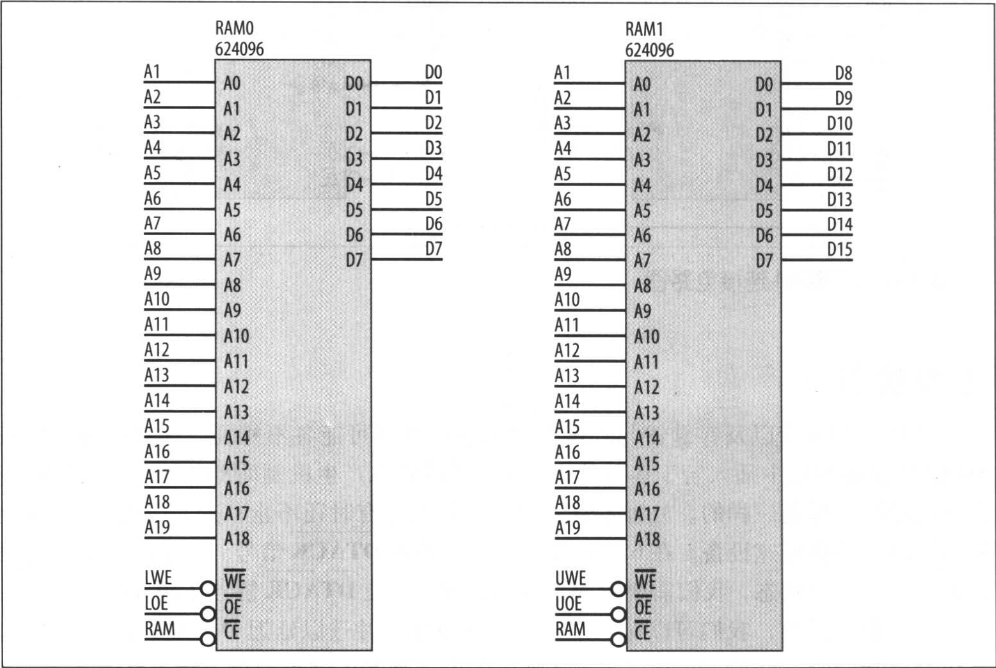


图 18-11：SRAM 连接电路图

ROM 的连接也和 SRAM 类似，如图 18-12 所示。

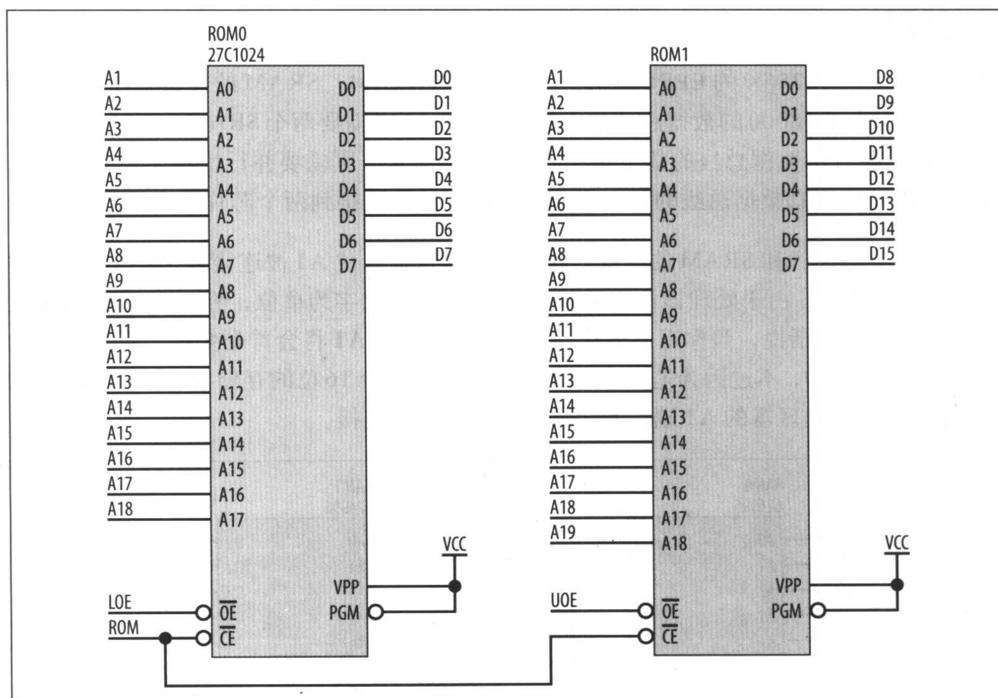


图 18-12：EPROM 连接电路图

等待状态

由于处理器的速度以及存储器与外围芯片的访问时间可能各不相同，你也许有必要在68000的存储周期中插入若干等待状态。这种等待状态产生机制同那些处理器支持异步存储周期的道理是一样的。处理器将发出一个输入（有时还不止一个），这会延长存储周期以响应那些慢速设备。在68000中，输入的则是 **DTACK** 信号。为了给一个给定设备插入一个等待状态，我们需要检测设备的访问，并使 **DTACK** 信号在额外的时钟周期内失效。换句话说，我们可以对一个给定的设备使用片选以延迟 **DTACK** 信号的降低。完成这一功能的电路很简单，不过最好要在PAL或是其他的可编程逻辑电路内实现，这可以使等待状态的变化更加容易。等待状态产生器还包括一个D型触发器（注2）（如图18-13所示），每一个触发器都表示一个额外的传输确认延迟的时钟周期。

注 2： 触发器是一个在时钟脉冲的改变边沿馈送 D 输入到 Q 输出的逻辑元件。

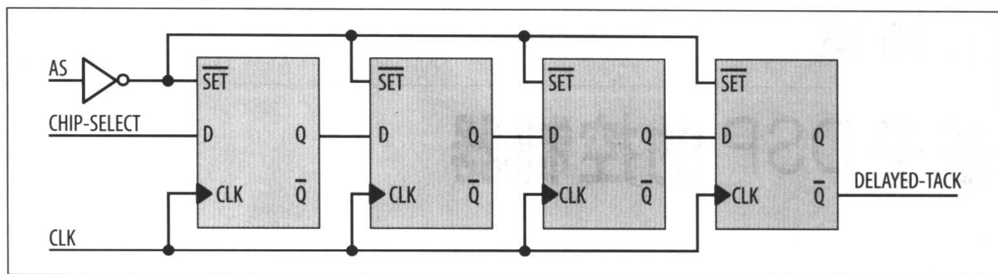


图 18-13：等待状态产生器

在存储周期之间，地址选通信号 **AS** 变为高电平。它首先被反转，然后被连接到触发器的每个低电平触发的**SET**输入端。因此，每个触发器的输出将在每个存储周期之间被置为高电平。这将会使它们从任何前一个周期中复位。地址解码器为这个特殊的外设产生一个片选信号，并连接到第一个触发器的D输入端。因此，对于每一个后继的时钟脉冲，由片选信号提供的 0 被触发器依次同步。经过 4 个时钟脉冲后，0 到达最后一个触发器的输出端Q，则触发器的反转输出端**Q**输出1。然后再由PAL输出到74LS05转换器，最后传送给处理器的 **DTACK** 信号。对于额外的等待状态，需要增加更多的触发器。对于那些需要不同等待状态的外设来说，需要使用它们自身的片选信号，以提供给第一个处理器的D输入端，然后选择所需的延迟级别。在产生统一的处理器确认信号之前，这些信号都必须复合各自的片选以激活/挂起输出。

大部分支持等待状态机制的处理器，目前都内置有可以通过软件配置的等待状态发生器，这就使得系统设计的任务变得更为简单。

下一章我们要介绍的是与上面不同类型的嵌入式处理器 —— 基于数字信号处理(Digital Signal Proassing，DSP）体系结构的处理器。

第19章

基于DSP的控制器

……世界的统一性，
表现于一切发生的事情都是相互联系的，
一切大大小小的事物无不受到时间的驱使。
……事物的统一性和必然顺序被打破了，
虽然打开的裂口很小，
可是由此这个世界的统一性还是被外来的新鲜事物所侵袭。
——赫曼 · 赫塞
《悉达多》

本章介绍另一种不同的处理器：DSP（数字信号处理器），DSP是用来执行算术密集型算法的专用处理器。第一款DSP处理器出现于20世纪80年代初，从那之后面向更多应用的处理器设备陆续被开发出来。这些处理器的主要特点在于，它们具有从内存或外设中快速读入读出数据的能力，而且它们的体系结构都已经被优化，以便能够对那些数据进行算术处理。

DSP的基本目的是要快速读入数据，并对这些数据执行复杂的算法，然后将运算的结果输出。很多DSP处理器有两个数据空间，即X和Y。它们能同时访问这两个数据空间，并且同时取回两个操作数进行处理。很多DSP处理器也是采用哈佛结构，因此有三个地址空间：一个是代码空间，另外两个是数据空间，它们都可以同时并发地存取。这些特性再加上技术上已经很成熟的ALU（算术逻辑单元），使DSP处理器具备了先进的数据处理能力。

DSP处理器通常应用于音频处理、视频或图像的处理、通信、雷达和声纳系统，以及生物医学方面。你的移动电话、DVD播放器，还有家庭影院中的环绕立体声扩音器，都内置有DSP。用耳蜗制成的仿生耳朵也使用了DSP芯片。

下面是 DSP 技术的一些应用实例：

- 汽车内的引擎控制和防滑刹车装置
- 数字广播和电视
- DVD 播放器和家庭影院
- 电子音乐合成器
- GPS 导航装置
- 雷达处理和声纳处理
- 飞机导航装置、飞船电子设备及导弹制导系统
- 工业用电机控制
- 机器人技术
- 虚拟现实系统
- 图像的处理、压缩和增强
- 模式识别和机器视觉
- 自适应滤波、快速傅立叶变换（FFT）及希尔伯特（Hilbert）变换
- 科学数据处理
- 医疗诊断装置、超声波诊断和医疗成像系统
- 移动电话、寻呼机、调制解调器、移动电话的基站、数字传真机
- 数字加密
- 数字专用自动交换分机（PABX）和非对称用户数字环线（ADSL）
- 回波（重影）消除
- 通信中展频处理
- 视频会议系统
- 说话人确认
- 语音的增强和识别
- 语音合成和编码
- 语音邮件系统

然而，这些应用只不过是 DSP 的开始。

目前生产DSP芯片的厂商有三个，它们分别为：德州仪器公司（Texas Instruments，TI）（*http://www.ti.com*），产品为TMS320系列；Analog Devices公司（*http://www.analogdevices.com*），产品为21xx和SHARC（21xxx）处理器；Freescale Semiconductor公司（*http://www.motorola.com*），产品为DSP56xxx处理器以及高端的MSC8100 StarCore处理器，这款高端处理器的设计目标是用来进行网络间的通信和信息的处理。还有很多其他的制造商正在把DSP功能芯片加入到它们的嵌入式控制器中。其中典型的例子就是Microchip公司（*http://www.microchip.com*）生产的dsPIC处理器。

德州仪器公司所生产的DSP范围很广，从小型、低成本DSP到超级计算机芯片。它所生产的TMS320C6000系列DSP使你的普通PC看起来像是一个锈迹斑斑的算盘。这一系列DSP是128位VLIW（Very Long Instruction Word，超长指令字）处理器，每一时钟周期内可以执行高达8条指令；能够以高达2000MIPS和900MFLOPS的速度运行，并且德州仪器仍在使这些处理器运行得更快。这些处理器主要设计用于复杂数据的关键计算（并且，如果你想体验一下，所需的花费可不少）。

德州仪器公司和Analog Devices所设计的DSP芯片，被用来构建大规模的并行DSP计算机。其中Analog Devices公司生产的SHARC芯片在单个机器上既能支持基于消息传递的MIMD（Multiple Instrument Multiple Data，多指令多数据）模式，也能支持共享存储器的MIMD模式。在共享存储器的并行计算机系统中，一个节点可以有6个SHARC处理芯片，这样一个节点可以和6个其他的节点进行消息传递。

或许现在你开始觉得一个SHARC比CRAY-1超级计算机有着更强的处理能力，那么我们可以说一个并行的SHARC机器却是立在你案头的可怕的怪物（但是在你对这种想法变得狂热起来之前，我们还是不打算设计这么一款机器。因为对你而言，这种机器太复杂、太昂贵了，而且所涉及的内容很多都超出本书的范围）。

Freescale Semiconductor公司的DSP56000芯片是一款24位的处理器，虽然被应用在很多领域，但其设计的主要目的却是用于音频方面。之所以选择24位的体系结构，是因为在音频处理方面24比特的字长是最经常采用的。仿生耳朵中就使用了DSP56000。

目前，DSP处理器不仅在其专用领域如信号处理方面做得很好，而且在通用控制领域也能工作得不错。相对于传统的处理器而言，一个装有DSP芯片的嵌入式系统在执行一些复杂软件和高级算法方面有着更高的效率。早期的嵌入式DSP系统通常采用DSP与微控制器复合的方式，其中DSP负责数据的处理而微处理器则执行一些通用的功能。虽然DSP在数据的处理方面很理想，但在其他的一些应用中表现的却不是特别的好。然而在同一个系统内放置两个处理芯片并不是最好的选择，因此对它的一个逻辑上的改进可使其成为一个具有混和功能的处理器：即把微处理器的一些功能和DSP芯片内核相结合。

到目前为止，DSP芯片的开发商已经研制出多种DSP体系结构，其中很多种都是基于嵌入式应用。它们把很多微处理器常用的子系统（如UART、SPI、ADC等）都集成到DSP的内核中。同时它们的指令集也是DSP指令（执行数据的移动和运算）和传统微处理器指令的混合。这种处理器对于以下一些应用都是很理想的：电机控制（尤其是在机器人技术方面）、神经网络和模糊控制、数据压缩、数字通信、数码相机，以及任何只需小硬件（相对廉价）的计算密集型应用。

在这一章，我们将要看到的是Freescale Semiconductor公司的DSP568000系列的DSP控制器，其中主要讨论的是DSP56805处理器芯片。本章主要就是介绍如何基于这种芯来构建计算机。DSP56800处理器芯片通常应用于小规模和低成本的嵌入式系统中，其主要功能是用来实现高级数字控制和处理。德州仪器公司和Analog Devices公司也生产类似的处理器，虽然在体系结构方面有些不同，但用这些芯片构建一个计算机的技术基本上是一样的。

DSP56800系列

与传统24位体系结构的DSP56000处理器不同，DSP56800系列处理器的为16位体系结构，这使得它更好适合小规模控制应用程序。虽然它们只能处理定点数（即整数），但对大部分的控制应用来说这就足够了。必要时，浮点算术运算也可以用软件的方法来实现。

这种体系结构的实现基于四个功能单元，每一个单元都有自己的寄存器，能够独立地执行操作，而且还可以和其他的几个单元并行执行。这四种功能单元为：程序控制器(program controller)，负责软件的执行；地址生成单元（Address Generation Unit, AGU)，用以处理总线的访问；数据ALU，执行算术运算；位操作单元(bit-manipulation unit)，用以有效而快速地执行位运算。

由于这些单元能够彼此独立地执行操作，这使得系统能够以很快的速度去执行软件，而且效率也很高。当指令执行时，数据ALU或是位操作单元能够执行相应的操作。同时，地址生成单元产生下一条指令的地址，而这时程序控制器将取回下一步要执行的另一条指令。而且，系统的指令集能够直接支持并行化。为了实现系统内的高吞吐量，处理器使用了多总线结构，其中包括三个内部地址总线，四个内部数据总线（其中三个用于内存和CPU，一个用于外设）。来自内存的两个操作数能够执行单个指令。这使得这种结构的机器在系统时钟频率为80MHz时，能够获得40MIPS的吞吐量。这意味着使用类似CISC（复杂指令集计算机）的指令集结构，能够获得类似RISC（精简指令集计算机）的性能。换句话说，这的确令人感到很满足了。

DSP56800系列芯片使用DO和REP指令，用硬件来完成循环指令的执行。DO指令允许定义一个任意大小的代码段，然后让处理器在硬件上执行该循环代码段。而且在每个循环执行一次后，不需要用计数器和条件分支指令去判断下一步该执行哪一条指令，这样就节省了处理器的开销。REP指令能够实现单个指令的执行循环，也能够在DO语句中嵌套执行。这样，不必花费什么开销就能得到多种循环形式，因而循环指令在DSP上的执行速度是非常快的。

DSP56800内核的编程模型如图19-1所示。

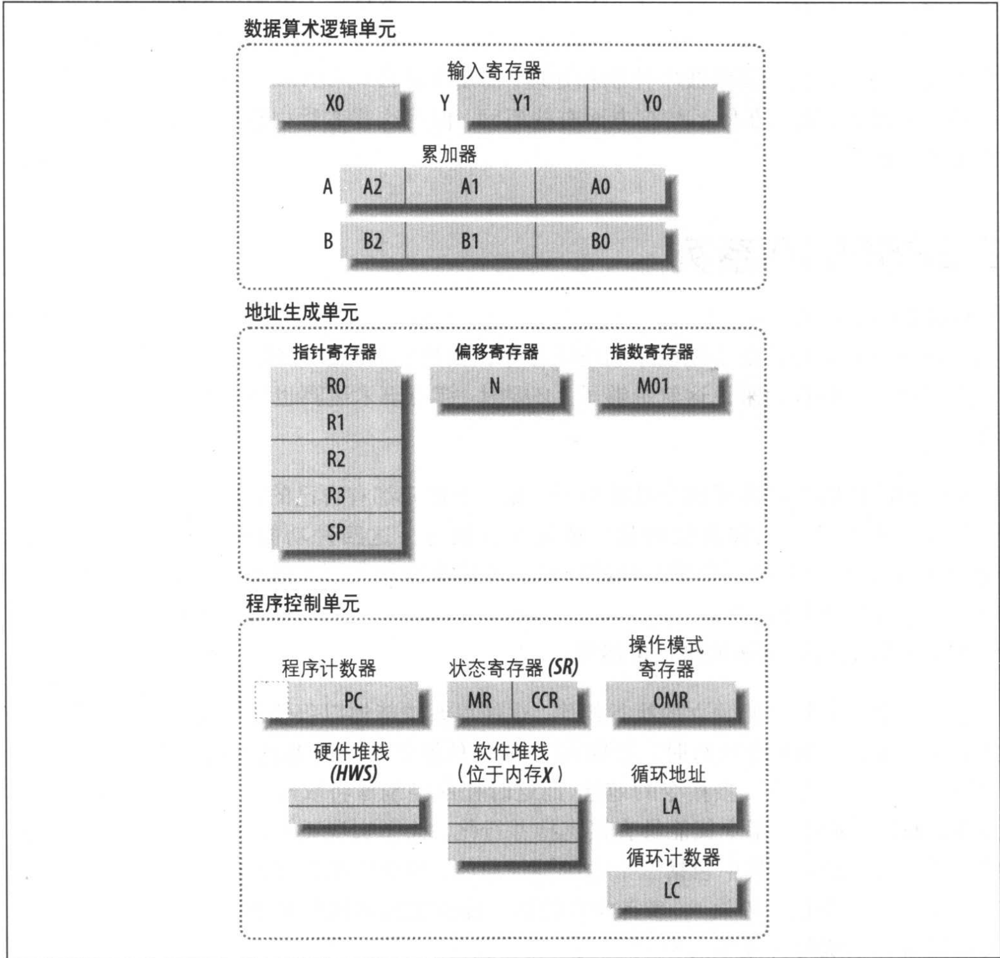


图19-1：DSP56800编程模型

DSP56800处理器包括两个36位的累加器、一个16×16位的乘法/累加处理单元(Multiply and Accumulate，MAC)，以及一个 16 位的桶形移位器。MAC 使你只用一条指令就可以将两个数的乘积加入到累积的结果中。MAC还能确保许多信号处理算法以及神经模糊程序的有效执行。内部的桶形移位器能够保证在单个周期内完成高达16位的向左或是向右的移位。然而，对于传统的处理器而言，如果你想将一个操作数向左移动15位，通常有两种选择：或者执行15个左移操作指令；或者通过一个循环体来实现，而且每个循环中要包括一条左移指令、一个计数器变量以及一个循环测试条件。同其他的许多DSP处理器一样，DSP56800 也能够在一个时钟周期内完成这一操作。

简而言之，DSP56800 具有紧凑而且高效的代码，使大部分的功能都能在相当短的时间内完成。通过使用这种处理器，可以很容易地设计出一种功能强大的嵌入式计算机系统。

下面来看如何使用DSP56805 处理器来构建一个系统，其中 DSP56805 是 DSP56800 系列中的一种，它主要用在工业控制方面。DSP56805 内部组成部分包括：一个 1K的可编程 RAM、一个用于存放引导装入程序 4K 的 ROM（用于从外部存储装置或是外围设备装载引导程序）、一个 63K 的程序闪存、一个 8K 的数据闪存和一个 4K 的数据 RAM。DSP56805 还具有外部数据总线和地址总线，这样使得处理器的存储容量能够很容易突破内部资源的限制。它的地址空间为 64K×16 位，能够访问 128KB 的外存储器。

注意：还有一些DSP568xx处理器有比这更大的地址空间。DSP5685x 系列能够支持最大为2M的程序存储器和最大为 8M 的数据存储器。

DSP56800处理器还能提供独立的程序及数据空间，从而使外部地址空间加倍。此外，处理器还有一个可编程的等待状态产生器，用以简化与外设间的连接。当处理器要访问一个给定的设备时，产生器可以编程提供 0、4、8 或 12 等待状态。

DSP56800 系列处理器通常都带有一组内置的外围设备，包括一个或两个串行外设接口(SPI)、若干个 16 位的通用定时器、一个监视定时器（被称为计算机适当操作 COP，由 Freescale Semiconductor 公司生产）、一个用于实时操作的定时器、一个用于访问音频编码译码器（由模 / 数转换器和数 / 模转换器构成）或其他 DSP 芯片的同步串行接口(Synchronous Serial Interface，SSI)，以及一些通用 I/O 总线。为了处理电机控制及相关应用，DSP56805 芯片还增加了两个 6 通道的脉宽调制单元（简称 PWM）、两个 4 通道的模 / 数转换器（其中每个通道为 12 位），还有两个用于测量电机位置的积分译码器。它还有一个 CAN 网络模块、两个串行接口（即串行通信接口，简称 SCI，Freescale Semiconductor 公司开发），以及 14 个专用 I/O 总线、18 个共享 I/O 总线。

DSP56805处理器的正常工作电压为3.0V~3.6V，但它的最大输入电压能达到5V，这使得它能很容易地连接多种外设（而对于其他的一些DSP56800系列处理器，根据使用芯片的不同，工作电压可以为4.57V~5.5V）。DSP56805处理器还有多种可选的低功耗和休眠模式，便于使用电池组供电的系统使用。

所有的DSP56800处理器都采用了JTAG端口，系统通过这种端口来连接专用的调试工具。JTAG端口也允许对处理器的板上闪存程序存储器的直接访问，这使得下载新代码的工作变得简单、快捷。

DSP56805处理器的结构框图如图19-2所示。

总而言之，DSP56805是一款很不错的处理器，接下来我们要看的就是如何用它来构造一个系统。为了简单起见，我们将其划分为若干个子系统来依次讨论。

基于DSP56805的计算机

DSP56805有9个电源引脚，其中每一个引脚都必须对地接一个100nF的陶瓷电容，用于退耦。每个电容的位置应当与它们各自的电源引脚尽可能地近。由于该处理器执行速度较高，因而会产生大量的噪音，所以四层电路板的构建是其首选（详见第6章）。对于所有设计，任何无用的输入都必须置为无效。

振荡器

和其他所有的处理器一样，DSP56805也需要一个时钟信号。而振荡器能够产生处理器执行指令操作所需的时钟频率：高可达80MHz（即每秒4千万条指令），低则只有几MHz（当然这是出于节能的需要）。有时处理器的时钟甚至可以完全停止（即所谓的DC操作，意味着时钟不再有AC信号输入）以进一步节约功耗（这款处理器的姊妹产品DSP56801，其时钟信号则完全来自内部的振荡器，因而不需要外部时钟发生电路）。

这种处理器有一个内置的振荡器电路单元，只需要一个频率范围在4MHz~8MHz之间的外部晶体振荡器和一些支持组件（如图19-3所示）。处理器能够在软件的控制下，将来自晶体振荡器的较低的频率在系统内部合成高达40MHz~110MHz的时钟频率。需要指出的是，虽然时钟发生电路能够产生110MHz的时钟频率，但是处理器的指令执行速度却达不到这么高。所以将时钟频率控制在80MHz以下来协调处理器以及软件。

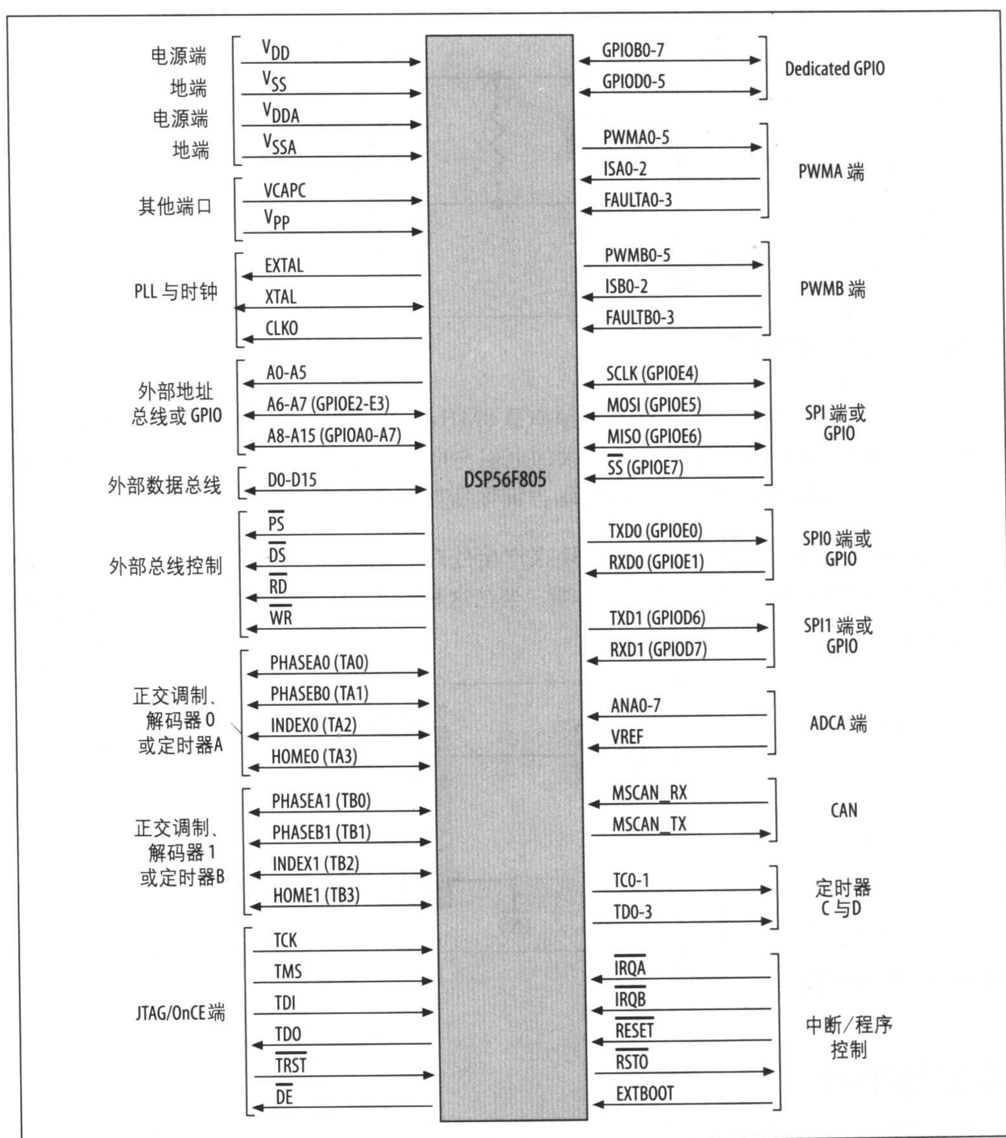


图 19-2：DSP56805 结构框图

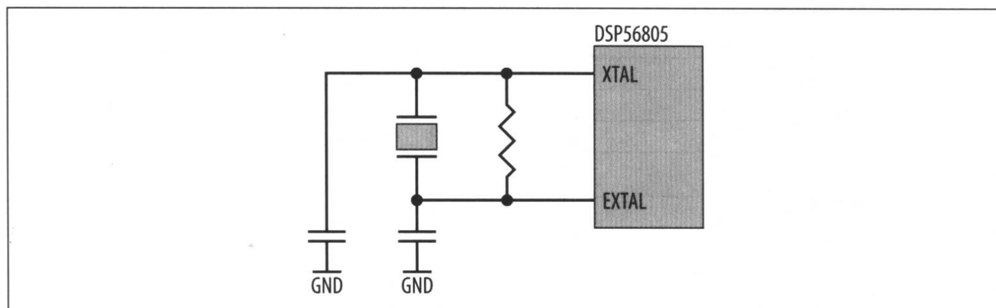


图 19-3：晶体振荡器电路

在一些典型的应用中，晶体振荡器的频率是 8MHz，其电阻值为 10MΩ，而退耦电容的大小约为15pF。不过晶体振荡器所要求的电阻与电容值可能是不同的，所以在选用前检查一下其制造商的技术参数。它将明确告诉你所使用的晶体振荡器的参数值。

或者你也可以选用一个外部振荡器模块来产生处理器的时钟（如图 19-4 所示）。而模块的输出被连接到处理器的 **XTAL** 输入端。当在这种配置的机器上执行操作时，**EXTAL** 必须接地。

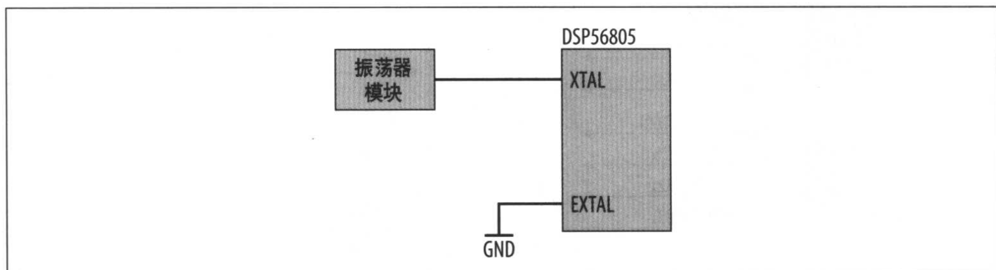


图 19-4：振荡器模块

复位与中断

DSP56805 有一个内部电源电路，用于正确地启动处理器。同时，处理器还有一个由内部定时器驱动的看门狗复位电路，用以将处理器从软件故障中恢复。因此，我们需要做的只是给系统添加一个外部复位装置，这样只需按一下按钮就可以手工重启机器了。对这样一个复位电路的要求是，它必须能够去除由于按钮按下产生的抖动（译注 1），确保

译注 1：当开关闭合时，开关器件在保持稳定之前会迅速地弹开几次，这样会产生误差，所以需要过滤掉由于弹开所产生的不稳定状态，这个过程便称为去抖动。

复位状态能够持续一个较短的时间。处理器将内部去抖动电路加于$\overline{\mathbf{RESET}}$输入之上，而且处理器还有一个电路用于保证复位状态能够持续一段合适的时间。因此我们的外部复位电路已经简化到只有一个按钮和一个上拉电阻（如图 19-5 所示）。还想再简化成什么样呢？

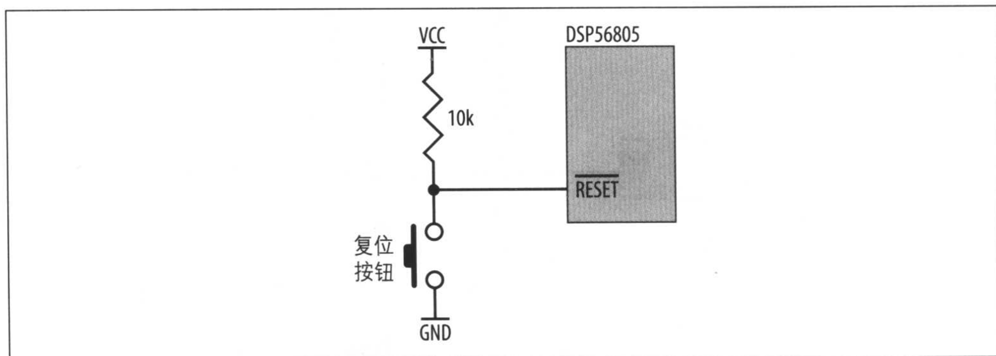


图 19-5：DSP56805 处理芯片上的外部复位装置

DSP56805处理器既可以从外部存储器中引导，在单芯片操作时也可以从内部ROM中引导启动。输入管脚 **EXTBOOT** 可以作为处理器执行复位的信号。如果 **EXTBOOT** 为低电平，则处理器执行来自内部 ROM 的代码。这就是所谓的 Mode 0 操作，包括两种形式：Mode 0A 模式与 Mode 0B 模式。Mode 0A 模式将所有的内存空间都映射到内部地址空间中，而 Mode 0B 模式则将低 32K 字（64KB）的地址空间映射到内部，高 32K 字的地址空间映射到外部。其中 Mode 0A 是默认的模式，Mode 0B 模式只能在软件控制下进入。

如果 **EXTBOOT** 管脚为高电平，那么处理器引导来自外部存储器的程序，这就是 Mode 3 操作（没有 Mode 1 或是 Mode 2 ，它们用于基于 ROM 的 DSP56800 处理器）。在操作执行时，处理器可以在软件的控制下从一种模式跳转到另外一种模式。

其他的DSP56800处理器也有类似的操作和内存映射模式，因此在具体应用时只需要参照技术手册的要求去做即可。

除了众多内部中断源（来自板上外设），DSP56805 芯片还有两个外部中断源：$\overline{\mathbf{IRQA}}$ 和 $\overline{\mathbf{IRQB}}$。外部接口设备（甚至整个外部系统）通过使用它们来请求处理器操作。不论外设是否被连接到外部中断源上，它们都需要一个外部上拉电阻。在图 19-6 所示的例子中，$\overline{\mathbf{IRQA}}$ 接有一个来自外设的中断源，而 $\overline{\mathbf{IRQB}}$ 未使用。

外部存储器

处理器通过一根16位的数据总线来访问外部程序存储器和外部数据存储器。数据和程序存储器能够在同一个存储器芯片共存，或者对同一存储器划分单独的程序地址空间和数

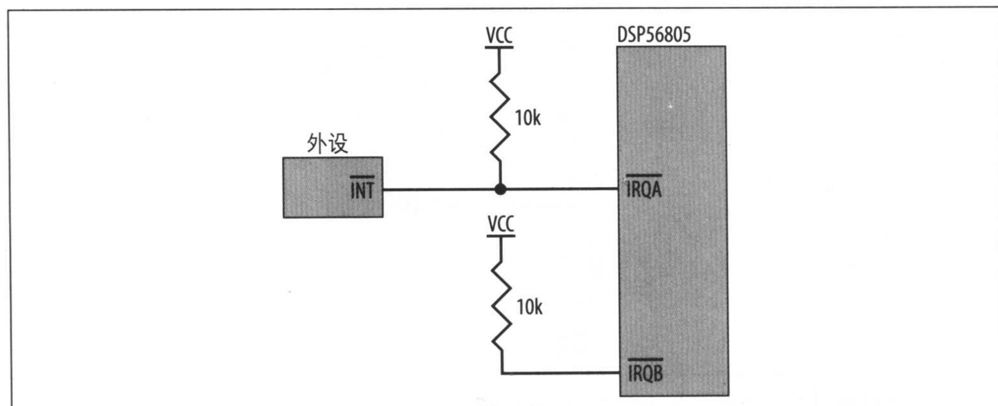


图 19-6：外部中断源

据地址空间。处理器有程序选通输出（program strobe，**PS**）和数据选通输出（data strobe，**DS**）两种不同的输出，用于表明访问内存的类型。

图 19-7 显示的是 DSP56805 处理器的读写周期的时序图。由于处理器有一个可编程的等待状态产生器，所以能够适应各种响应时间不同的外部存储器设备或外设的需要。

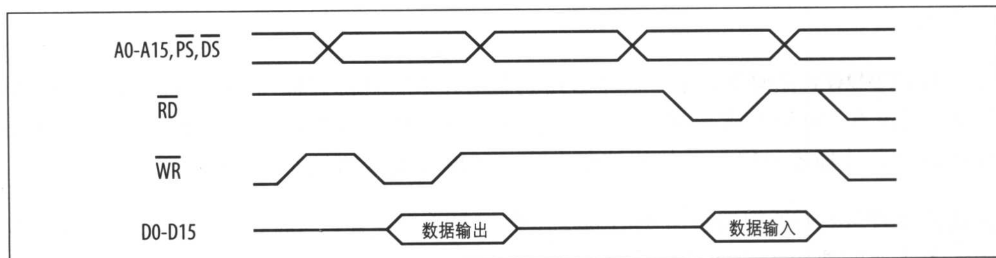


图 19-7：DSP56805 内存周期

DSP56805 处理器可以通过“无胶合”接口实现与内存之间的互连，这意味着不再需要其他的外部逻辑。图 19-8 所示是 DSP56805 与两个 64K 的程序 SRAM 之间的连接电路图。

当访问程序地址空间时，程序选通（**PS**）为低电平，因此它可以当作SRAM的片选（chip select）信号。类似地，数据存储器也可以采用同样的设置，这样数据选通（**DS**）便成

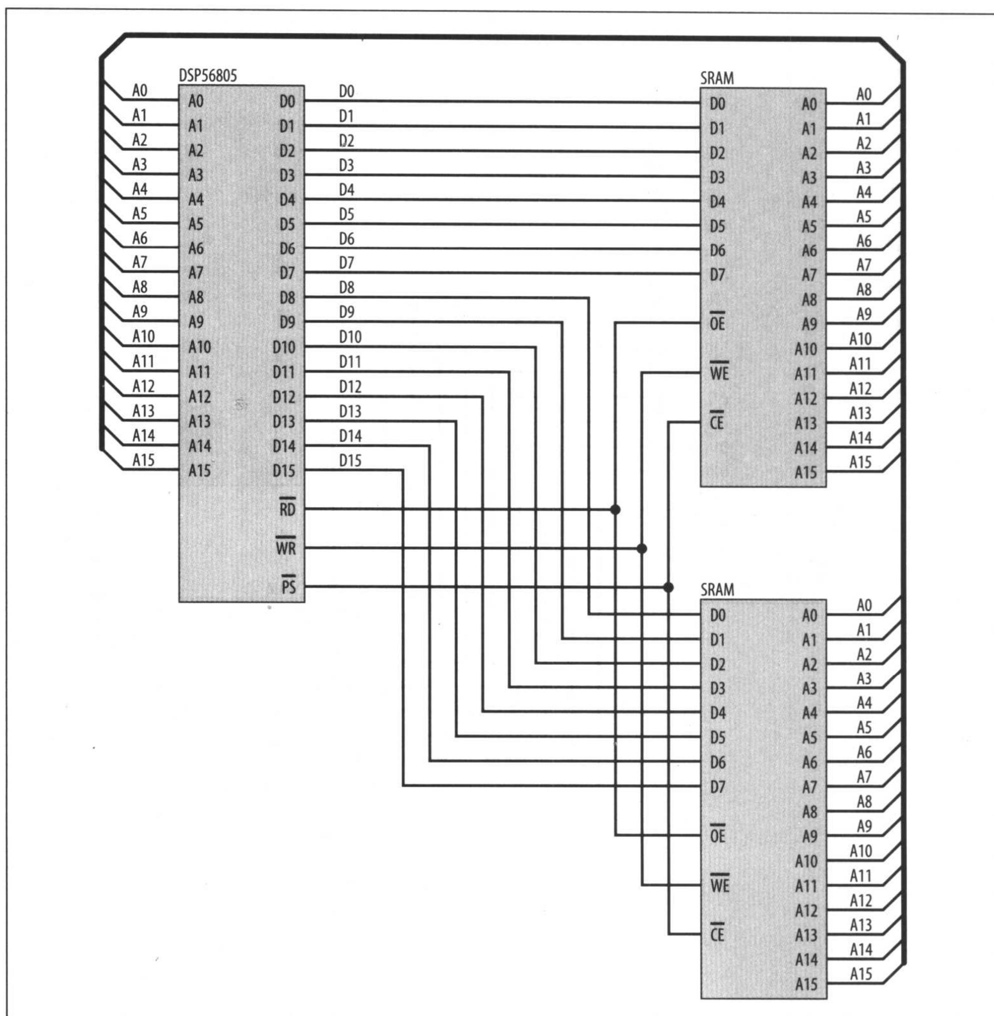


图 19-8：连接 DSP56805 与程序 SRAM

为数据存储器的片选信号（见图 19-9 所示）。需要指出的是，我们使用“程序存储器(memory)”或是“数据存储器”这两个词的目的仅是为了说明这些芯片的使用目的，而不是对存储器进行分类。因而，同一种类型的SRAM芯片就可以完全满足这两方面的需要。

因此，我们的DSP56805计算机共有四个SRAM芯片，被均匀地分配给程序存储器和数据存储器。每个存储区域的大小为 64K×16 字节（包括两个 19 位的存储器芯片），总共提供 128K 的程序空间和 128K 的数据空间。所以，系统的内存的总容量为 256K。如果

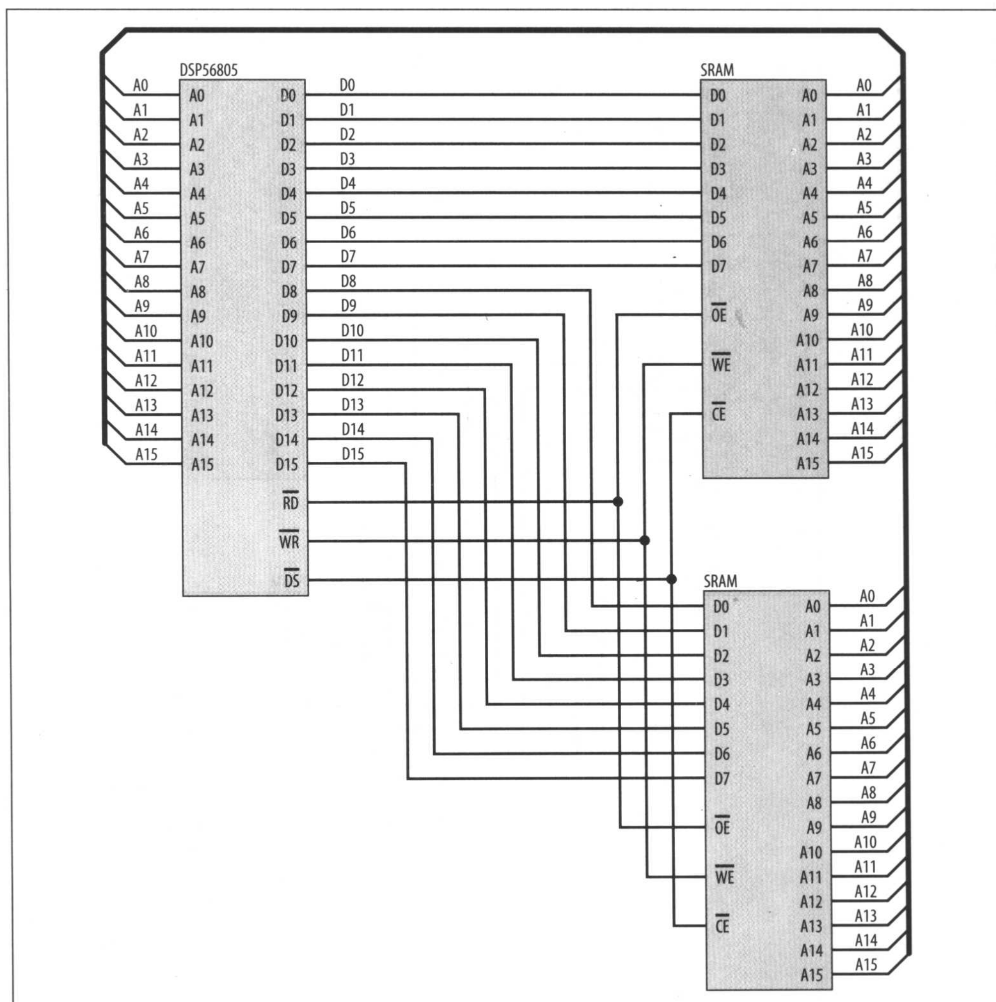


图 19-9：连接 DSP56805 与数据 SRAM

需要更多的存储空间，可以通过使用内存条来增加可用空间，如第 15 章所介绍的使用 AT90S8515。

值得注意的是，你没有必要一定使用独立的程序和数据空间，只要有两片 SRAM，它们完全可以同时存在于同一芯片之上（如图 19-10 所示）。

在这个例子中，程序选通（$\overline{\textbf{PS}}$）和数据选通（$\overline{\textbf{DS}}$）都被忽略了，因为我们的主要目的不再是区分数据和程序空间。SRAM 的芯片使能（$\overline{\textbf{CE}}$）输入端被接地，这使得这些设备能够一直处于有效状态。之所以接地是因为SRAM $\overline{\textbf{CE}}$为低电平时起作用，输出使能（$\overline{\textbf{OE}}$）

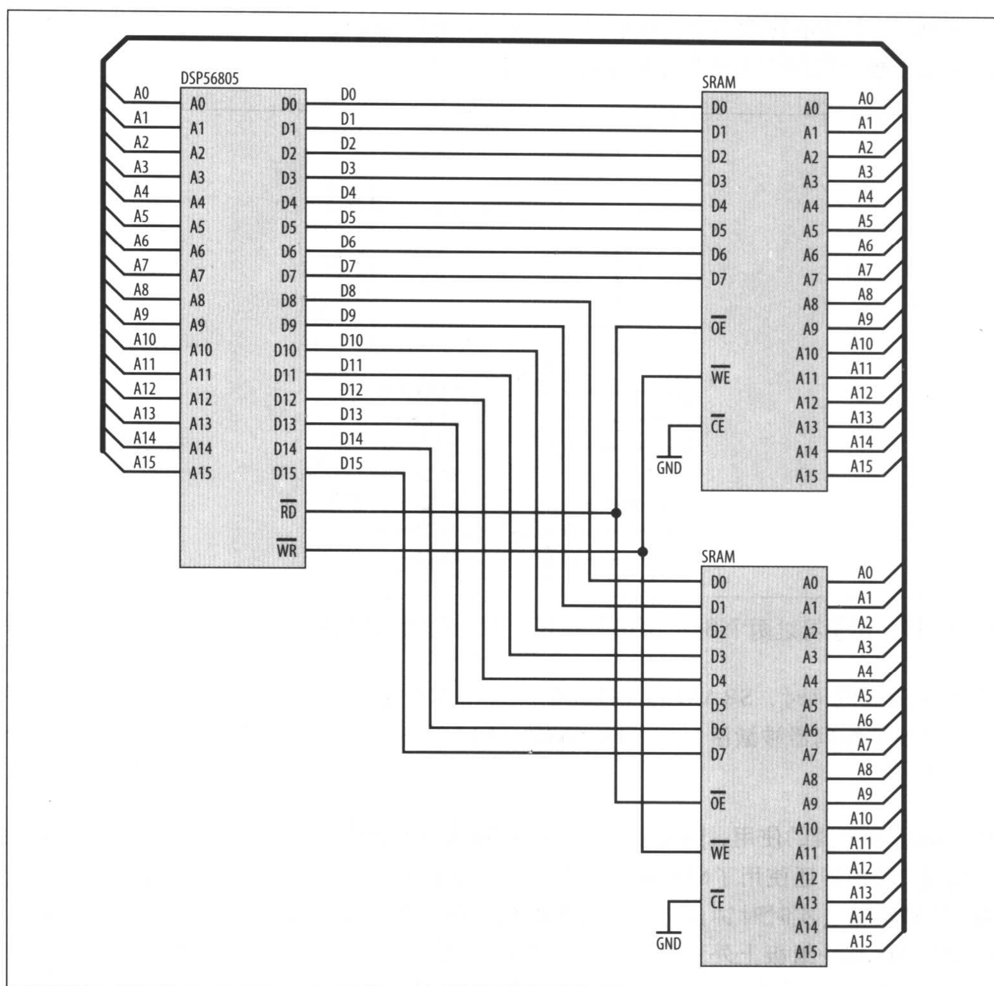


图 19-10：程序和数据共享存储器

和写使能（**WE**）也是如此。因而，在本例中输出使能或写使能将激活SRAM芯片。需要指出，永久性激活SRAM将会增加系统的功耗。当然，也可以复合**DS**和**PS**，于是它们中的任何一个为低电平都可以激活SRAM芯片，但是这样做需要增加额外的逻辑电路，而且也没有这个必要。

如果在你的内存空间中存在不同的组成部分，例如一个更小的数据SRAM和一些外设，那么必须用数据选通（**DS**）来作为SRAM和外设的使能信号。最合理的方法是，使用**DS**来激活地址解码器，从而选择相应的设备。而且需要强调的是，必须使用**DS**来访问外围设备，因为你不可能离开外围设备而直接执行代码！

在图 19-11 中显示的是一个地址解码器的例子，它既可以选择两个 32K SRAM，也可以选择位于数据空间中的 8 个外围设备中的任何一个。

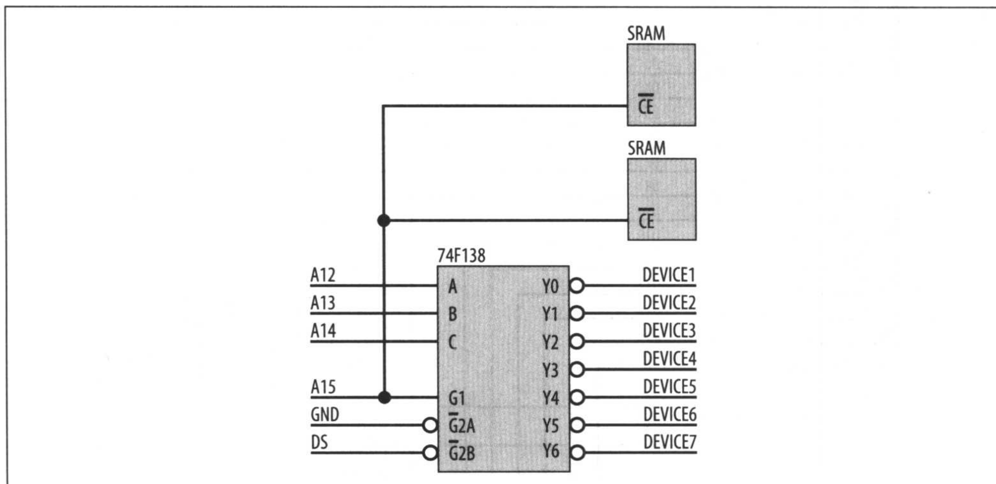


图 19-11：用于寻址两个 32K SRAM 和 8 个外设的地址解码器

当 **A15** 为低电平时，SRAM 芯片被选中。当它为高电平，且数据选通（**$\overline{\text{DS}}$**）为低电平时，则地址解码器被激活；而 **A12**、**A13**、**A14** 的不同状态决定了 8 个外设中的哪一个被选中。

由于地址解码器的使用，你可以在系统中同时使用 8 个基于总线的外设。为了可扩展性的需要，处理器还使用了 SPI 接口，SPI 接口可以使你增加额外的设备，如模/数转换器、数/模转换器、实时时钟日历、非易失性数据存储器以及其他的一些设备主机。当然，DSP56805 那一组板上外设已经使其功能异常强大了。DSP56805 处理器的 SPI、并行 I/O 以及串口接口的使用与小型微控制器的接口类似。DSP56805 拥有较多的片上外设，这也造就了这款性能极其强大的处理器。

JTAG

DSP56805 内部集成了一个 JTAG 端口，以协助系统的调试。JTAG 端口包括四个专用信号（见表 19-1）。

表19-1：JTAG信号

信号名	功能
TDI	测试数据输入
TDO	测试数据输出
TMS	测试模式选择
TCK	测试时钟

Freescale Semiconductor公司还向JTAG标准集中增加了一些信号，其中$\overline{\textbf{TRST}}$（Test Reset，复位测试）用来复位JTAG的状态机，$\overline{\textbf{OE}}$（Debug Event，调试事件）的作用等同于一个中断输出，用以显示发生在片上仿真器（On-Chip Emulation，OnCE）模块上一个事件（诸如一个断点信息等）的发生。

虽然JTAG主要用于调试系统，但是由于通过它可以实现对处理器内部的完全控制，所以它也可以用来对内部程序闪存进行再编程。Freescale Semiconductor公司制定了DSP56F80x系列处理器片上闪存的编程规范（AN1935/D）。我们能从该公司的主页上得到使用JTAG/OnCE接口的DSP56F80x系列芯片的详细信息，其中包括处理过程中所遇到的所有细节问题和实例，以及实例的源代码。

Freescale Semiconductor公司的软件开发包基于CodeWarrior C编译器，它为DSP56800系列处理器提供了软硬件开发工具。

Hardware

O'REILLY®

嵌入式硬件设计



嵌入式计算机系统与我们的日常生活息息相关，它们可能就隐匿在我们的移动电话、PDA、汽车、电视、电冰箱、空调以及其他许多设备中。事实上，嵌入式系统是当今计算机工业成长最快速的部分之一。

随着适用嵌入式计算机系统的设备的增多，程序员、业余爱好者以及各类工程师对如何设计和构建自己的设备越来越有兴趣。此外，本书对这些计算机系统所提供的基础知识对于需要评估和应用这些系统的任何人而言都会大有裨益。

本书作为第二版加入了新一代处理器和微控制器（包括最新的MAXQ处理器）的相关信息。本书不仅为初学者提供了嵌入式设计的基本知识，也为高级系统设计者提供了有用的参考资料。

目前市面上可以看到的相关书籍，不是专门探讨如何为特定微处理器编写程序的，就是只强调嵌入式系统的设计原理而不提供任何实用信息，唯有本书做到了两者兼顾。本书的作者John Catsoulis具有丰富的实践经验，他会告诉你如何设计和构建全新的嵌入式设备与计算机化的小器件，以及如何修改和扩展一个现有的系统。

除了实际的实例，本书还提供了许多提示和警告，帮你避免各种错误和陷阱。本书的内容涵盖：

- 嵌入式系统的理论和实践
- 理解电路原理图和技术手册
- 为嵌入式系统供电
- 制作并调试嵌入式系统
- 诸如PIC、Atmel的AVR以及Motorola 68000系列的处理器
- 数字信号处理（DSP）体系结构
- 用于添加外部设备的协议：SPI和I^2C
- RS-232C、RS-422、红外通信以及USB
- CAN和Ethernet
- 脉宽调制（PWM）与电机控制

如果你想要构建自己的嵌入式系统，或是修改现有的系统，本书将会向你提供所需的知识和实践技能。

ISBN 978-7-5083-5372-2



O'Reilly Media, Inc. 授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

定价：39.80元